9.4

Developing Applications for IBM MQ

Note

Before using this information and the product it supports, read the information in “Notices” on page 1249.

This edition applies to version 9 release 4 of IBM® MQ and to all subsequent releases and modifications until otherwise indicated in new editions.

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

Contents

Developing applications for IBM MQ

You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

New to developing applications for IBM MQ?

To learn about developing applications for IBM MQ, visit IBM Developer:

- IBM MQ Developer Essentials *(learn the basics, run a demo, code an app, take more advanced tutorials)*
- IBM MQ Downloads for Developers *(including free developer editions and trial versions)*

You might also find it easier to develop your applications if you are familiar with the concepts described in the following sections:

- "Application development concepts" on page 6
- "Design considerations for IBM MQ applications" on page 47

Support for object oriented languages and frameworks

IBM MQ provides core support for applications developed in the following languages and frameworks:

- JMS
- Java
- C++
- .NET

See also "Object-oriented applications" on page 15.

.NET supports applications developed in many languages. To illustrate using the IBM MQ classes for .NET to access IBM MQ queues, the MQ product documentation contains information for the following languages:

- C# example code and sample applications
- C++ sample applications
- Visual Basic sample applications

See "Writing and deploying IBM MQ .NET programs" on page 559.

IBM MQ supports .NET Core for applications in Windows environments from IBM MQ 9.1.1 and for applications in Linux® environments from IBM MQ 9.1.2. For more information, see "Installing IBM MQ classes for .NET" on page 537.

ALW IBM MQ also supports the AMQP clients that implement the OASIS AMQP 1.0 protocol.

MQ Light, Apache Qpid clients like Apache Qpid Proton and Apache Qpid JMS APIs are based on this protocol.

The MQ Light APIs are available at IBM MQ Light.

The Apache Qpid clients are available at QPid Proton.

The following language bindings are provided as-is:

- a Go binding
- a JavaScript API implementation that works with Node.js applications

Support for programmatic REST APIs

IBM MQ provides support for the following programmatic REST APIs to send and receive messages:

- IBM MQ messaging REST API
- z/OS IBM z/OS Connect EE
- IBM Integration Bus
- IBM DataPower® Gateway

See “Developing REST applications with IBM MQ” on page 679, and also the tutorial Get started with the IBM MQ messaging REST API in the IBM MQ area of IBM Developer. This tutorial includes examples in the following languages, provided as-is, for use with the IBM MQ messaging REST API:

- Go example that uses the MQ messaging REST API
- Node.js example using HTTPS module
- Node.js example with Promise module

Support for procedural programming languages

IBM MQ provides support for applications developed in the following procedural programming languages:

- C
- Windows Visual Basic (Windows systems only)
- COBOL
- z/OS Assembler (IBM MQ for z/OS only)
- z/OS PL/I (IBM MQ for z/OS only)
- IBM i RPG (IBM MQ for IBM i only)

These languages use the message queue interface (MQI) to access message queuing services. See “Developing MQI applications with IBM MQ” on page 692. Note that the IBM MQ Object Model, used by the object oriented languages and frameworks, provides additional functions that are not available to the procedural languages using the MQI.

Specifying the application name

ALW

Before IBM MQ 9.1.2, you could specify an application name on Java or JMS client applications. From IBM MQ 9.1.2, you can also specify the application name on additional programming languages. For more information, see “Specifying the application name in supported programming languages” on page 50.

Related tasks

“Developing applications for MQ Telemetry” on page 1196

“Developing Microsoft Windows Communication Foundation applications with IBM MQ” on page 1218
The Microsoft Windows Communication Foundation (WCF) custom channel for IBM MQ sends and receives messages between WCF clients and services.

Related reference

“Developing applications for Managed File Transfer” on page 1175
Specify programs to run with Managed File Transfer, use Apache Ant with Managed File Transfer, customise Managed File Transfer with user exits, and control Managed File Transfer by putting messages on the agent command queue.

Application development concepts

You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

For information about the types of application you can write for IBM MQ, see “Developing applications for IBM MQ” on page 5 and “Actions that your applications can perform” on page 7.

Related concepts

"Design considerations for IBM MQ applications" on page 47
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 IBM MQ.

Actions that your applications can perform

You can develop applications to send and receive messages that you need to support your business processes. You can also develop applications to manage your queue managers and related resources.

Actions that your applications can perform on IBM MQ for Multiplatforms

On Multiplatforms, you can write applications that perform the following actions:

- 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 IBM MQ platforms.
- Use message queuing from within CICS® for the following systems:
  - 

TXSeries® for AIX®
  - 

IBM i
  - 

Windows
- Use message queuing from within Encina for the folllowing systems:
  - 

AIX
  - 

Windows
- Use message queuing from within Tuxedo for the following systems:
  - 

AIX
  - AT&T
  - 

Windows
- Use IBM MQ as a transaction manager, coordinating updates made by external resource managers within IBM 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 IBM MQ environment, or run from an IBM MQ client environment.

Actions that your applications can perform on IBM MQ for z/OS

On z/OS, you can write applications that perform the following actions:

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

- IBM 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.
- IBM 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 topics such as "Building z/OS batch applications" on page 985 and "z/OS batch considerations" on page 706 for z/OS batch programs.
- Although multiple instances of IBM 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.
- IBM 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 Shared queues and queue sharing groups.

z/OS The differences between the supported environments, and their limitations, are explained further in "Using and writing applications on IBM MQ for z/OS" on page 857.

Related concepts

"Application development concepts" on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

"Design considerations for IBM MQ applications" on page 47
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 IBM MQ.

"Writing a procedural application for queuing" on page 696
Use this information to learn about writing queuing applications, connecting to and disconnecting from a queue manager, publish/subscribe, and opening and closing objects.

"Writing client procedural applications" on page 879
What you need to know to write client applications on IBM MQ using a procedural language.

"Using IBM MQ classes for JMS/Jakarta Messaging" on page 79
IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

"Using IBM MQ classes for Java" on page 335

Use IBM MQ in a Java environment. IBM MQ classes for Java allow a Java application to connect to IBM MQ as an IBM MQ client, or connect directly to an IBM MQ queue manager.

"Developing C++ applications" on page 508
IBM MQ provides C++ classes equivalent to IBM MQ objects and some additional classes equivalent to the array data types. It provides a number of features not available through the MQI.

"Building a procedural application" on page 962
You can write an IBM MQ application in one of several procedural languages, and run the application on several different platforms.

Related tasks

"Using the IBM MQ sample procedural programs" on page 1018
These sample programs are written in procedural languages, and demonstrate typical uses of the Message Queue Interface (MQI). IBM MQ programs on different platforms.

"Developing .NET applications" on page 535
IBM MQ classes for .NET allow .NET applications to connect to IBM MQ as an IBM MQ MQI client or to connect directly to an IBM MQ server.

"Developing Microsoft Windows Communication Foundation applications with IBM MQ" on page 1218
The Microsoft Windows Communication Foundation (WCF) custom channel for IBM MQ sends and receives messages between WCF clients and services.

Securing

Multi Applications, application names, and application instances

Before you start to design and write your applications, familiarize yourself with the basic concepts of applications, application names, and application instances.

Applications

Multi

Connections to a queue manager are considered to be from the same *application* if they provide the same *application name*. The application name is displayed as the APPLTAG attribute of the DISPLAY CONN(*) TYPE CONN command.

Notes:

1. For applications using a version of the IBM MQ client earlier than IBM MQ 9.1.2, the application name is automatically set by the IBM MQ client. Its value depends on the application programming language, and the platform the application is running on. See PutApplName for more information.
2. For IBM MQ client applications using an IBM MQ client at IBM MQ 9.1.2 or later, it is possible to set the application name to a specific value. In most cases, this does not require changes to application code or a need to recompile the application. See "Using the application name in supported programming languages" on page 51 for further information.

Application instances

Multi

Connections are further subdivided into *application instances*. An instance of an application is a set of closely related connections which provide one 'unit of execution' for that application. Typically, this is a single operating system process, which can have a number of threads and associated IBM MQ connections.

On IBM MQ for Multiplatforms an application instance is associated with a specific Connection Tag. The queue manager automatically associates new connections with an existing application instance, when it can see that they are related.

Notes:

- If using client connections, these processes might connect to the queue manager over one or more running channels.
- In JMS applications, an application instance maps to a specific JMS connection and all associated JMS sessions.

Application instances are particularly important on IBM MQ for Multiplatforms when using uniform cluster automatic application balancing. On IBM MQ for Multiplatforms platforms, you can view currently connected application instances using the DISPLAY APSTATUS command.

In some cases, the queue manager cannot correctly perform connection to application instance association, in particular:

- If multiple connections are made on a shared conversation from the same process, using different application names.
- If older level client libraries are in use. For example, IBM MQ JMS client installations at IBM MQ 9.1.2 and earlier.

In these situations, if the applications do not define themselves as reconnectable, this will be permitted, but some of the application instance groupings might be incorrect. If any of the connections are declared as MQCNO_RECONNECT then this significantly negatively affects application balancing; the MQCONN call will, therefore, be rejected with MQCNO_RECONNECT_INCOMPATIBLE.

Related concepts

"Specifying the application name in supported programming languages" on page 50
Before IBM MQ 9.2.0, you could already specify an application name on Java or JMS client applications. From IBM MQ 9.2.0 this feature is extended to other programming languages on IBM MQ for Multiplatforms.

Application programs using the MQI

IBM MQ application programs need certain objects before they can run successfully.

Figure 1 on page 10 shows an application that removes messages from a queue, processes them, and then sends some results to another queue on the same queue manager.

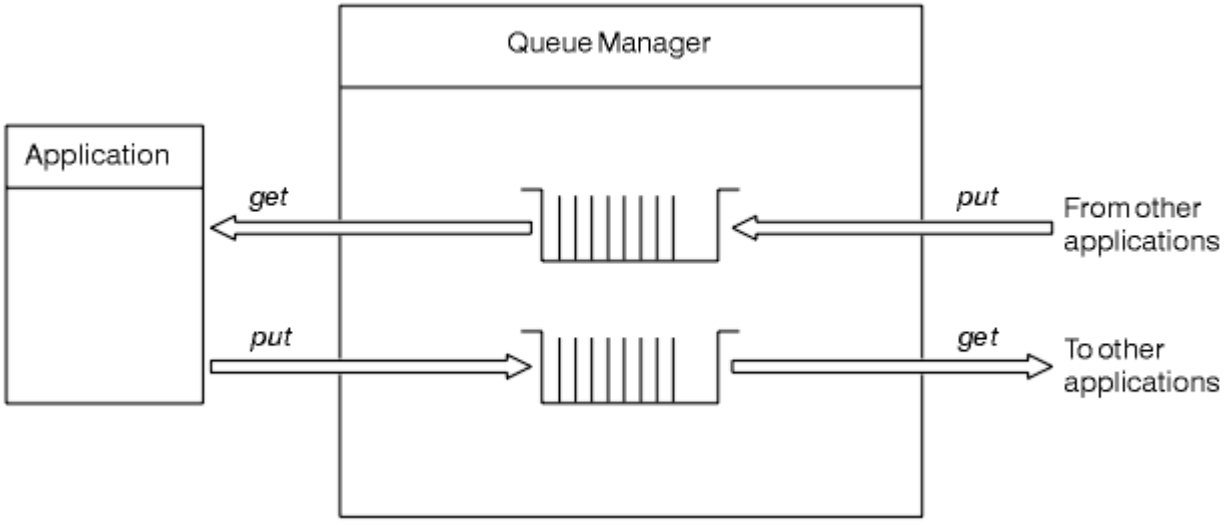


Figure 1. Queues, messages, and applications

Whereas applications can put messages onto local or remote queues (using MQPUT), they can only get messages directly from local queues (using MQGET).

Before this application can run, the following conditions must be satisfied:

- The queue manager must exist and be running.
- The first application queue, from which the messages are to be removed, must be defined.

- The second queue, on which the application puts the messages, must also be defined.
- The application must be able to connect to the queue manager. To do this it must be linked to IBM MQ. See “Building a procedural application” on page 962.
- The applications that put the messages on the first queue must also connect to a queue manager. If they are remote, they must also be set up with transmission queues and channels. This part of the system is not shown in Figure 1 on page 10.

Using client connections to connect to multiple IBM MQ queue managers

It is possible to configure client connected applications to connect to more than one queue manager (for load balancing or service availability reasons).

The primary mechanisms to achieve this in the IBM MQ client are use of client channel definition tables, see Configuring client channel definition tables, or connection lists.

It is also possible to achieve similar behavior using external load balancing products or by wrapping IBM MQ connection code in a 'stub' which can redirect host names or IP addresses.

Each of these techniques comes with some restrictions, and might be more or less suitable to particular application requirements. The following sections, although not exhaustive, describe particular aspects you should consider, and the effect of these different approaches on these aspects.

IBM MQ uniform clusters, see About uniform clusters, provide a powerful mechanism to achieve horizontal scaling of applications across multiple queue managers building on the basic mechanism of the CCDT to provide multiple destinations. Uniform clusters can provide capabilities beyond what is possible using an external load balancer unaware of the underlying IBM MQ protocols, and avoid some of the problems discussed below, therefore, consider using a uniform cluster in preference to other techniques where applicable.

Attention: You should use with caution applications using IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging, including those using one of the IBM MQ Resource Adapters, that connect to queue managers using load balancing technologies. If you encounter problems, recreate those problems without attempting to use load balancing.

There are multiple issues involved, all of which mean that such connections are at best problematic and at worst wholly unreliable:

- Particular care is required when connecting any application which makes multiple connections to the queue manager using any form of load balancing. This includes all applications using the IBM MQ Classes for JMS/Jakarta Messaging as these create multiple IBM MQ connections in general usage. If using an external load balancer or custom code stub, this must route connections from the same application instance to the same queue manager at all times.
- Use of XA transaction management or JTA (Java Transaction API) relies on the ability to connect consistently to the same queue manager – in practice this is unlikely to ever be practical with any form of load balancing.
- - Uniform cluster management relies on being able to instruct clients to reconnect to specific queue managers without interference. It is not advisable to attempt to combine external load balancing with use of Uniform Clusters

You should use the IBM MQ uniform cluster functionality to achieve horizontal scaling of applications across multiple queue managers, rather than external load balancing technologies. See Configuring a uniform cluster, and the following topics, for information on uniform clusters, including how you create and use uniform clusters.

Terms used in this information

CCDT- multi-QMGR

Means a CCDT file that contains multiple client connection (CLNTCONN) channels with the same group, that is the queue manager name client connection (QMNAME CLNTCONN) attribute, where different CLNTCONN entries resolve to different queue managers.

This is distinct from a CCDT file that contains multiple CLNTCONN entries that are simply different IP addresses or host names for the same multi-instance queue manager, which is an approach you might choose to combine with a code stub.

If you do choose a CCDT multi queue manager approach, you need to choose whether to prioritize the entries or have randomized work load management (WLM):

Prioritized
Use multiple alphabetically ordered entries with CLNTWGHT(1) and AFFINITY(PREFERRED) attributes to remember the last good connection.

Randomized
Use CLNTWGHT(1) and AFFINITY(NONE) attributes. You can adjust the WLM weighting across differently scaled IBM MQ servers by adjusting the CLNTWGHT

Note: You should avoid large differences in CLNTWGHT between channels.

Load balancer
Means a network appliance with a Virtual IP address (VIP) configured with port monitoring of the TCP/IP listeners of multiple IBM MQ queue managers. How the VIP is configured in the network appliance depends on the network appliance you are using.

The following choices relate only to applications sending messages, or initiating synchronous request and reply messaging. The considerations for applications servicing those messages and requests, for example, the listeners are completely separate, and discussed in detail in "Connecting a message listener to a queue".

Scale of code change required for existing applications that connect to a single queue manager

CONNAME list, CCDT multi-QMGR, and Load balancer
MQCONN("QMNAME") to MQCONN("*QMNAME")

The queue manager name might be in the Java Naming and Directory Interface (JNDI) configuration for Java Platform, Enterprise Edition (Java EE) applications. Otherwise this requires a one character code-change.

Code stub
Replace existing JMS or MQI connection logic with a code stub.

Support for different WLM strategies

CONNAME list
Prioritized only.

This is likely to have a negative effect on the code.

CCDT multi-QMGR
Prioritized or random.

This is not likely to have any effect on the code.

Load balancer
Any, including each connection for all messages.

This is likely to have a positive effect on the code.

Code stub
Any, including each message for all messages.

This is likely to have a positive effect on the code.

Performance overhead while primary queue manager is unavailable

CONNAME list
Always tries first in list.

This is likely to have a negative effect on the code.

CCDT multi-QMGR
Remembers last good connection.

This is likely to have a positive effect on the code.

Load balancer
Port monitoring avoids bad queue managers.

This is likely to have a positive effect on the code.

Code stub
Can remember last good connection, and retry intelligently.

This is likely to have a positive effect on the code.

XA transaction support

CONNAME list, CCDT multi-QMGR, and Load balancer
The transaction manager needs to store recovery information that reconnects to the same queue manager resource.

An MQCONN call that resolves to different queue managers generally invalidates this. For example, in Java EE, a single connection factory should resolve to a single queue manager when using XA.

This is likely to have a negative effect on the code.

Code stub
Code stub can meet the XA requirements for a transaction manager, for example, multiple connection factories.

This is likely to have a positive effect on the code.

Admin flexibility to hide infrastructure changes from apps

CONNAME list
DNS only.

This is likely to have a negative effect on the code.

CCDT multi-QMGR
DNS and shared file-system, or shared file-system, or CCDT file push.

This is not likely to have any effect on the code.

Load balancer
Dynamic virtual IP address (VIP).

This is likely to have a positive effect on the code.

Code stub
DNS or single queue manager CCDT entries.

This is not likely to have any effect on the code.

Avoiding disruption around planned maintenance

There is another situation that you need to consider and plan for, which is how to avoid disruption to applications, for example, errors and timeouts visible to the end users, during planned maintenance of a queue manager. The best approach to avoid disruption is to remove all work from a queue manager before it is stopped.

Consider a request and reply scenario. You want all in-flight requests to complete, and the replies to be processed by the application, but you do not want any additional work to be submitted into the system. Simply quiescing the queue manager does not fulfill this need, as well-coded applications receive a return code RC2161 MQRC_Q_MGR_QUIESCING exception, before they receive their reply messages for in-flight requests.

You can set PUT(DISABLED) on the request queues used to submit work, while leaving the reply queues both PUT(ENABLED) and GET(ENABLED). In this way, you can monitor the depth of the request,

transmission, and reply queues. Once they all stabilize, that is, in-flight requests complete or time out, you can stop the queue manager.

However, good coding in the requesting applications is required to handle a PUT(DISABLED) request queue, which results in the return code RC2051 MQRC_PUT_INHIBITED error, when trying to send a message.

Note that the exception does not occur when creating the connection to IBM MQ, or opening the request queue. The exception occurs only when an attempt is made to actually send a message, using the MQPUT call.

Building a code stub that includes this error handling logic for request and reply scenarios, and asking your application teams to use such a code stub in the future, can help you develop applications with consistent behavior.

Developing flexible and scalable client applications

For fault tolerance and scalability, deploying client applications which support connections options into uniform clusters allows the instances of the application to be rebalanced between queue managers.

See About uniform clusters for an overview of uniform clusters.

Ideally, this re-balancing is invisible to the application, but only certain types of application are suitable for this type of deployment, and some consideration might be needed in the application design.

These considerations fall into two main categories:

- Rare *error paths* which might exist already for re-connectable applications, but become more likely when deployed into a uniform cluster. For example, following a reconnect, any inflight unit of work is backed out, and browse cursors reset. These might be a rare event for your re-connectable application in its current environment and, therefore, not handled as cleanly as possible by the application code. Reviewing application logic, to ensure appropriate handling is in place for such situations, helps avoid unexpected issues arising.
- *Affinities* to a particular queue manager. If you know that an application must always connect back to the same or a specific queue manager, the application should be configured to reconnect to that queue manager, or not have its connection to that queue manager enabled. However, these affinities might be temporary, such as waiting for a response message. Influencing the balancing algorithm to account for these affinities from the application code is discussed in the following section. For more details on these options, and how to achieve a similar approach through configuration, rather than application code, see Influencing application re-balancing in uniform clusters.

Influencing reconnection options in the MQI

See Reconnection options for more information on MQCNO_RECONNECT.

If you know that an application must always connect back to the same or a specific queue manager, it should be configured as MQCNO_RECONNECT_Q_MGR or MQCNO_RECONNECT_DISABLED.

Influencing the balancing algorithm in the MQI

However, you might want to control or influence that re-balancing behavior to suit the needs of specific types of application; for example, minimizing interruptions to in flight transactions, or ensuring requester applications receive their responses before being moved.

Certain default desirable behaviors are assumed and discussed in Influencing application re-balancing in uniform clusters. You can also influence the behavior for specific applications at configuration or deployment time through the client.ini file as discussed in that topic.

In other situations, it might make more sense to make the balancing behavior and requirements part of the application logic. In these cases, the same relevant characteristics of the application can be supplied to IBM MQ when connecting to the queue manager on the MQCONNX call, in a structure called MQBNO (balancing options).

If you provide an MQBNO structure, it must supply all of the information required by IBM MQ to make a decision about how and when the application should be asked to reconnect to a different queue manager.

You must supply:

- The **Type** of application
- The **Timeout** at which the instance is rebalanced regardless of state
- Any special **BalanceOptions**

The exception to this, is that you can leave the timeout as MQBNO_TIMEOUT_DEFAULT if required. In this case, the timeout resolves to any value in the client.ini file, application, or global stanzas, if provided, and failing that, to the base default of 10 seconds.

See MQBNO for details on the format of this structure.

For .NET applications, see Influencing application re-balancing in .NET for further information.

Object-oriented applications

IBM MQ provides support for JMS, Java, C++, and .NET. These languages and frameworks use the IBM MQ Object Model, which provides classes that provide the same functionality as IBM MQ calls and structures.

Some of the languages and frameworks that use the IBM MQ Object Model provide additional functions that are not available when you use procedural languages with the message queue interface (MQI).

For details of the classes, methods and properties provided by this model, see "The IBM MQ Object Model" on page 16.

JMS

IBM MQ provides classes that implement the Jakarta Messaging 3.0 and Java Message Service 2.0 specifications. For details of IBM MQ classes for JMS, see Using IBM MQ classes for JMS. For information about the differences between IBM MQ classes for Java and IBM MQ classes for JMS, to help you decide which to use, see "Developing JMS/Jakarta Messaging and Java applications" on page 78.

IBM MQ Message Service Client (XMS) for C/C++ and IBM MQ Message Service Client (XMS) for .NET provide an application programming interface (API) called XMS that has the same set of interfaces as the Java Message Service (JMS) API. For more information, see "Developing XMS .NET applications" on page 594.

Java

See Using IBM MQ classes for Java for information about coding programs using the IBM MQ Object Model in Java.

Stabilized IBM will make no further enhancements to the IBM MQ classes for Java and they are functionally stabilized at the level shipped in IBM MQ 8.0. For information about the differences between the IBM MQ classes for Java and the IBM MQ classes for JMS to help you decide which to use, see "Developing JMS/Jakarta Messaging and Java applications" on page 78.

C++

IBM MQ provides C++ classes equivalent to IBM MQ objects and some additional classes equivalent to the array data types. It provides a number of features not available through the MQI. See Using C++ for information about coding programs using the IBM MQ Object Model in C++. 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.

.NET

See Developing .NET applications for information about coding .NET programs using the IBM MQ .NET classes. 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.

Related concepts

"Developing MQI applications with IBM MQ" on page 692

IBM MQ provides support for C, Visual Basic, COBOL, Assembler, RPG, pTAL, and PL/I. These procedural languages use the message queue interface (MQI) to access message queuing services.

Technical overview

"Application development concepts" on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

Related reference

Developing applications reference

The IBM MQ Object Model

The IBM MQ Object Model consists of classes, methods and properties.

The IBM MQ Object Model consists of:

- *Classes* representing familiar IBM 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 IBM MQ objects.

When creating an IBM MQ application using the IBM MQ Object Model, you create instances of these classes in the application. 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 IBM 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.

Multi **MQDistributionList**
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.

Multi **MQDistributionListItem**
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 an IBM MQ program that uses the MQI, IBM MQ returns connection handles and object handles to the program.

These handles must be passed as parameters on subsequent IBM MQ calls. With the IBM 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.

IBM MQ messages

An IBM MQ message consists of message properties and application data. The message queuing message descriptor (MQMD) contains the control information that accompanies the application data when a message travels between the sending and receiving applications.

Parts of a message

IBM MQ messages consist of two parts:

- Message properties
- Application data

Figure 2 on page 18 represents a message and shows how it is logically divided into message properties and application data.

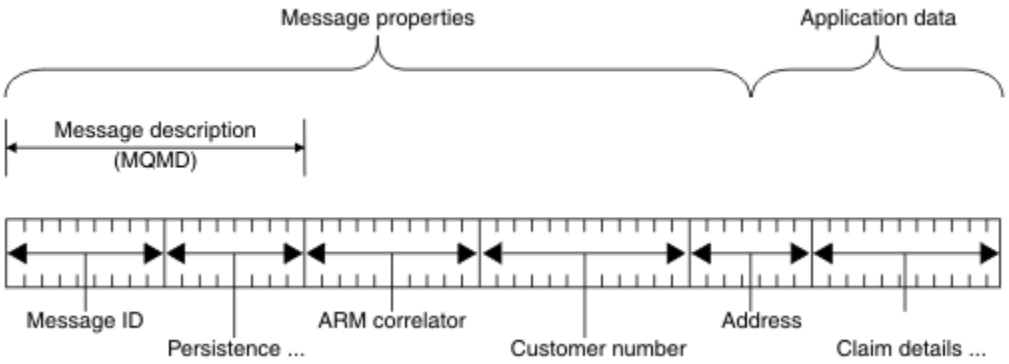


Figure 2. Representation of a message

The application data that is carried in an IBM MQ message is not changed by a queue manager unless data conversion is carried out on it. Also, IBM 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.

ALW On AIX, Linux, and Windows, the *MaxMsgLength* attribute of the queue manager and the queue defaults to 4 MB (4 194 304 bytes) which you can change up to a maximum of 100 MB (104 857 600 bytes) if required.

IBM i On IBM i, the *MaxMsgLength* attribute of the queue manager and the queue defaults to 4 MB (4 194 304 bytes) which you can change up to a maximum of 100 MB (104 857 600 bytes) if required. If you are intending to use IBM MQ messages greater than 15 MB on IBM i, see "Building your procedural application on IBM i" on page 968.

z/OS On 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. For more information, see "The data in your message" on page 731.

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 MQPUT and MQPUT1 for more information about these calls.

Message descriptor

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

For a full description of the MQMD structure, see MQMD - Message descriptor.

For a description of how to use the fields within the MQMD that contain information about the origin of the message, see "Message context" on page 45.

There are different versions of the message descriptor. Additional information for grouping and segmenting messages (see "Message groups" on page 42) is provided in Version 2 of the message descriptor (or the MQMDE). This is the same as the Version 1 message descriptor but has extra fields. These fields are described in the MQMDE - Message descriptor extension.

Types of message

There are four types of messages defined by IBM MQ.

These four messages are:

- Datagram
- Request messages
- Reply messages
- Report messages
  - Types of report message
  - Report message options

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 MsgType.

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 might use datagrams is one that displays flight information in an airport lounge. A message might contain the data for a whole screen of flight information. Such an application is unlikely to request an acknowledgment for a message because it probably does not matter if a message is not delivered. The application sends an update message after a short 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, if 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 is 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 do not need to check the settings of these flags. This is because these applications typically 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 about how to use report messages, see Report.

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. IBM MQ defines a range of feedback codes for applications to use.

For more information about feedback and reason codes, see Feedback.

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.)

Multi ***Reports and segmented messages***

If a message is segmented and you ask for reports to be generated, you might receive more reports than you would have done had the message not been segmented. Reports on segmented messages are only available on Multiplatforms.

For a description of segmented messages, see “Message segmentation” on page 764.

For reports generated by IBM 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 IBM 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 IBM 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 IBM 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 IBM 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 IBM MQ headers that you find.

Retrieving 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, you might get more than one report for a 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 things:

- 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 typically encoded according to the particular machine on which the queue manager is running.

When you put a message on a queue, you typically 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)
- IBM MQ defined messages, for example Programmable Command Formats

  IBM 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 Format.

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 947). 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 Options

Coded character sets

IBM 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, IBM 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, IBM 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. See “Writing data-conversion exits” on page 947 for information about invoking and writing data-conversion exits.

Message priorities

You can either set the priority of message to a numeric value, or let the message take the default priority of the queue.

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 afterward. 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 719), 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 IBM 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 IBM 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 IBM 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 IBM MQ application, or the other way round. To make a property available to a JMS application, assign it the prefix "usr"; it is then available (without the prefix) as a JMS message user property. For example, the IBM MQ property *usr.myproperty* (a character string) is accessible to a JMS application using the JMS call `message.getStringProperty('myproperty')`. Note that JMS applications are unable to access properties with the prefix "usr" if they contain two or more U+002E (".") characters. A property with no prefix and no U+002E (".") character is treated as if it had the prefix "usr". Conversely, a user property set in a JMS application can be accessed in an IBM MQ application by adding the "usr." prefix to the property name inquired on in an MQINQMP call.

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 an IBM 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 toward the length of the message for the queue and the queue manager, but they do count toward 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 for an application to successfully put a message with a buffer that is larger than the value of *MaxMsgLength*, if the buffer includes properties. This is because, even when represented as MQRFH2 elements, message properties do not count toward the length of the message. The MQRFH2 header fields add to the properties length only if one or more folders are contained and every folder in the header contains properties. If one or more folders are contained in the MQRFH2 header and any folder does not contain properties, the MQRFH2 header fields count toward the message length instead.

On an MQGET call, properties of the message do not count toward 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 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 contains the application message data to be sent and any properties represented in the message data.

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 IBM 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, IBM 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.

IBM MQ property names use character U+002E (.) for logical grouping of properties. This divides up the namespace for properties. Properties with the following prefixes, in any mixture of lowercase or uppercase are reserved for use by the product:

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

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" - reserved for use by IBM MQ classes for JMS.
   - "usr.JMS" - not valid.

   The only exceptions 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 IBM 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 IBM 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 name beginning " mq " in any mixture of lowercase or uppercase and not beginning "mq_usr" can contain only one "." character (U+002E). Multiple "." characters are not allowed in properties with those prefixes.
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 - Set message property 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 syntax shown.

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. In this case the MQMDE fields are treated as logically part of the MQMD from a properties perspective. See 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 property (MQBYTE[ ]) or character string property (MQCHAR[ ]) in that it 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.

Selectors

A message selector is a variable-length string used by an application to register its interest in only those messages that have properties that 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. See Code page conversion for a list of supported code pages.

Specifying a CCSID for which there is no IBM 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 on page 31 shows the process of selection using the MQSUB call.

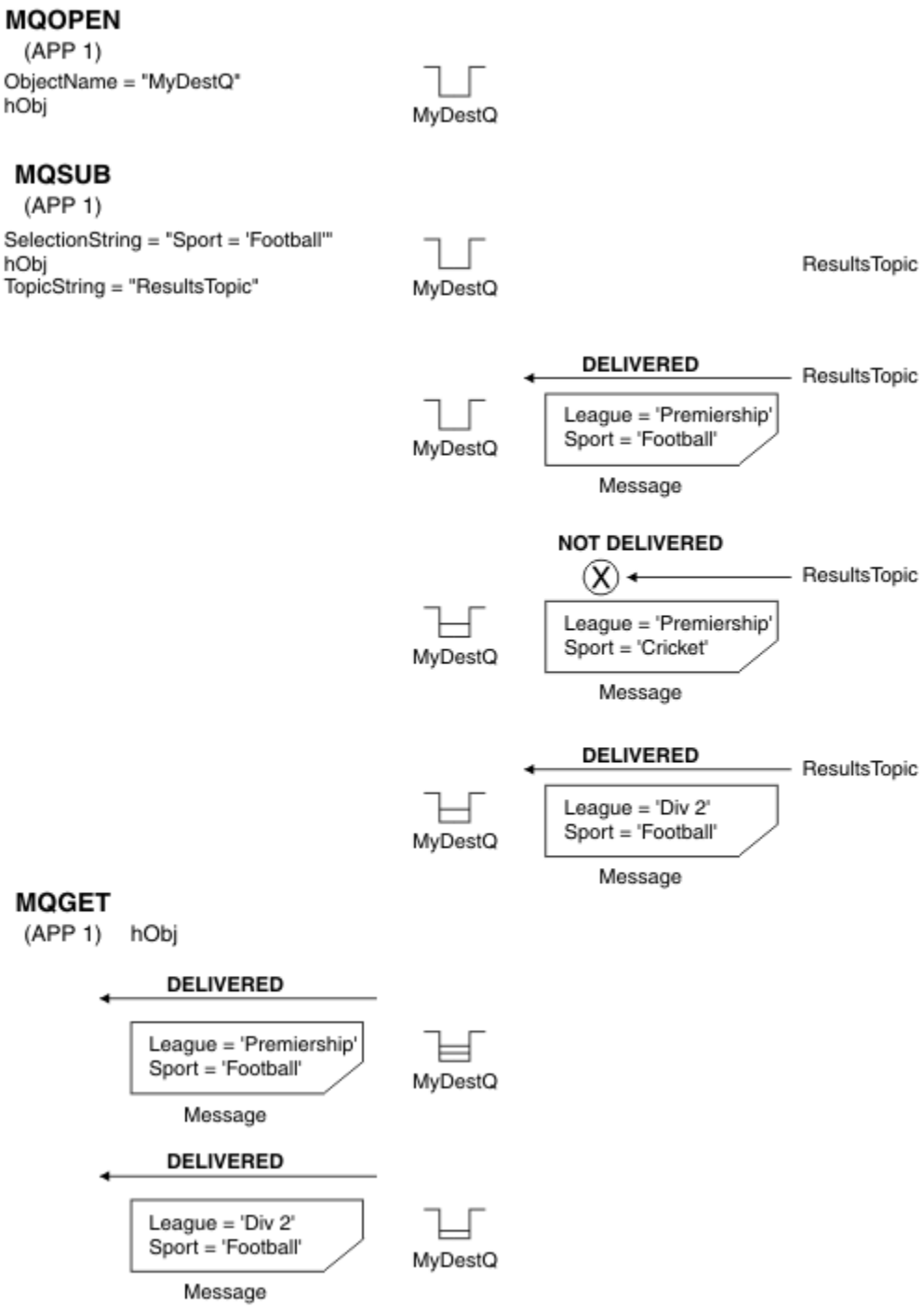


Figure 3. Selection using MQSUB call

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 32 shows the process of selection using the MQOPEN call.

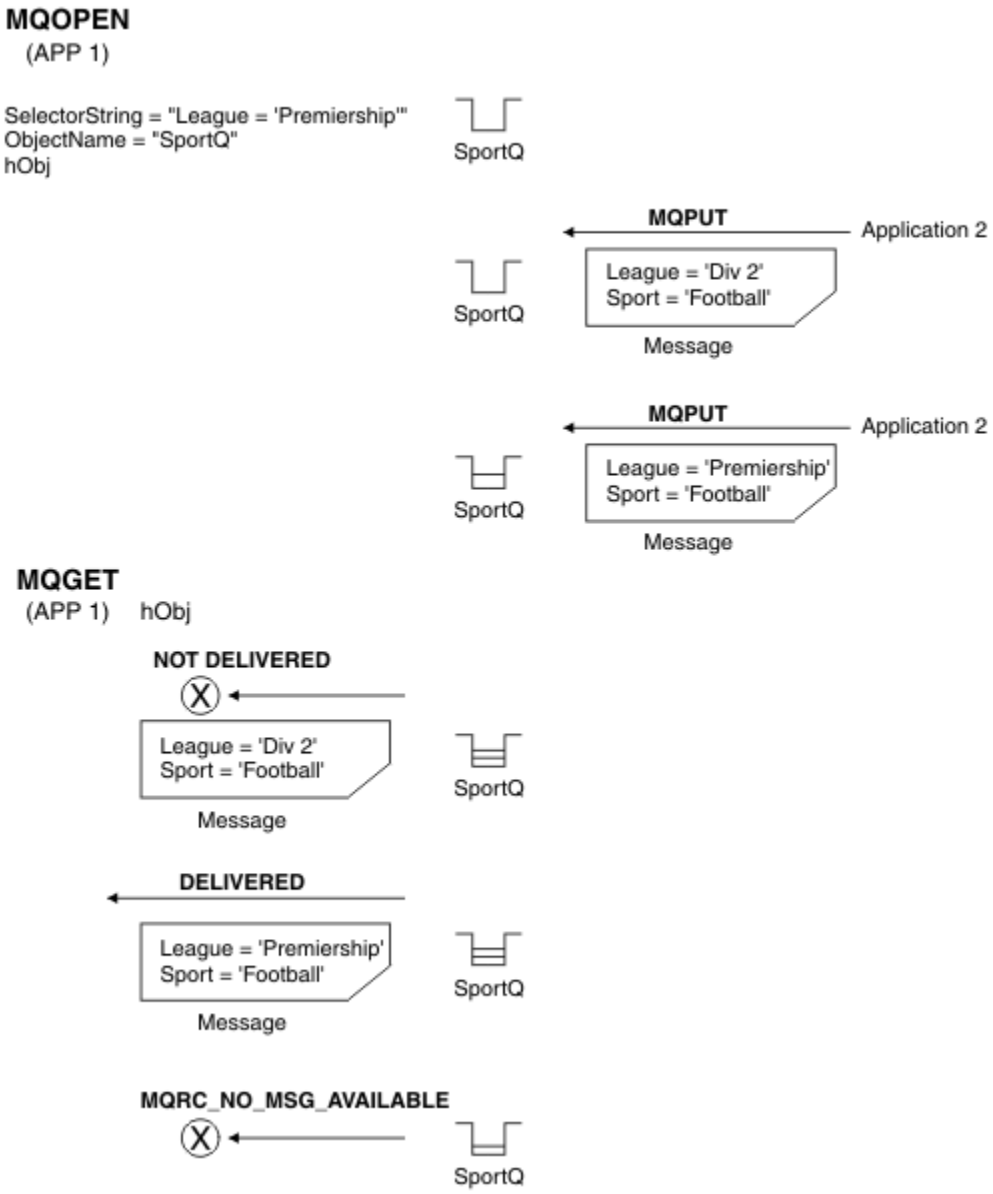


Figure 4. Selection using MQOPEN call

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 previous example 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.

Related concepts

"Selection string rules and restrictions" on page 39

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

Selection behavior

Overview of IBM 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. For example, it might be the case if the selection string is set to "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 can 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.

Selection can be performed only on the properties that are associated with a message, not the message itself, unless an extended message selection provider is available. Selection can be performed on the message payload only if an extended message selection provider is available.

Each message property has a type associated with it. 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. For example, 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. The subscriber does not receive that message to its destination queue.

Messaging performance

Selecting messages from a queue requires IBM MQ to sequentially inspect each message on the queue. Messages are inspected until a message is found that matches the selection criteria or there are no more messages to examine. Therefore, messaging performance suffers if message selection is used on deep queues.

To optimize message selection on deep queues when selection is based on JMSCorrelationID or JMSMessageID, use a selection string of the form:

- JMSCorrelationID ='ID:*correlation_id*'
- JMSMessageID='ID:*message_id*'

where:

- *correlation_id* is a String containing a standard IBM MQ correlation identifier.
- *message_id* is a String containing a standard IBM MQ message identifier.

Note: The selector should only reference one of the properties. Using a selector that has one of these formats offers a significant improvement in performance when selecting on JMSCorrelationID and offers a marginal performance improvement for JMSMessageID. For more information, see "Message selectors in JMS" on page 139.

Using complex selectors

Selectors can contain many components, for example:

a and b or c and d or e and `f` or g and h or `i` and `j`... or y and z

Use of such complex selectors can have serious performance implications and excessive resource requirements. As such, IBM MQ will protect the system by failing to process overly complex selectors that could result in a system resource shortage. Protection can occur on selection strings that contain more than 100 tests, or when IBM MQ detects that the limit on the size of the operating system stack is being approached. You should thoroughly try and test the use of selection strings with many components, on the appropriate platforms, to ensure that the protection limits are not reached.

The performance and complexity of selectors can be improved by simplifying them using additional parenthesis to combine components. For example:

(a and b or c and d) or (e and `f` or g and h) or (`i` and `j`) ...

Related concepts

"Selection string rules and restrictions" on page 39
Familiarize yourself with these rules about how selection strings are interpreted and character restrictions to avoid potential problems when using selectors.

Message selector syntax

An IBM MQ message selector is a string with syntax that 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 uppercase; however, they are not case-sensitive.

If the selector is provided via the API, IBM 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, and an extended message selection provider is not available, `MQRC_SELECTION_NOT_AVAILABLE` is returned to the application. If the syntax of the selection string is incorrect or a property name is not valid when a subscription is resumed, a `MQRC_SELECTOR_SYNTAX_ERROR` is returned to the application. If property name validation was disabled when the property was set (by setting `MQCMHO_NONE` instead of `MQCMHO_VALIDATE`) and an application subsequently puts a message with in invalid property name, this message is never selected.

No error is returned at the time the selector is presented if IBM MQ determines that an administratively defined subscription selector is using extended message syntax, as indicated by the **`DISPLAY SUB`** parameter **SELTYPE** having the value `EXTENDED`. In this case, the checking of the syntax of the selection string is deferred until publish time (see `MQRC_SELECTION_NOT_AVAILABLE`).

A selector can contain:

- Literals:

- String literals are enclosed in single quotation marks. Two consecutive single quotation marks represent a single quotation mark. Examples are 'literal' and 'literal''s'. Like Java string literals, these use the Unicode character encoding. You cannot use double quotation marks to enclose a string literal. Any sequence of bytes can be used between the single quotation marks.
- A byte string is one or more pair of hexadecimal characters enclosed in double quotation marks and prefixed by `0x`. Examples are `"0x2F1C"` or `"0XD43A"`. The length of a byte string must be at least one byte. If a selector byte string is matched to a message property of type `MQTYPE_BYTE_STRING`, no special action is taken on leading or trailing zero. The bytes are treated as another character. Endianness is also not considered. The length of both selector and property byte strings must be equal, and the sequence of bytes must be the same.

  Examples of byte string selections (assume *myBytes* = `0AFC23`) which match are:

  - `"myBytes = "0x0AFC23"" = TRUE`

  The following string selections do not match:

  - `"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 uppercase 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 uppercase or lowercase L; this does not affect how the number is stored or interpreted. IBM 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`. IBM 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 uppercase 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 a D character (not case-sensitive). 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 27 and “Property name restrictions” on page 28 for more information.

Note: Selection can be performed on the message payload only if an extended message selection provider is available.

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. IBM MQ maps the recognized JMS field and property names to the message property values. See "Message selectors in JMS" on page 139 for more information. As an example, the selection string "JMSPriority >=" selects on the Pri property found in the jms folder of the current message.

- Overflow/underflow:

  For both decimal and approximate numeric numbers, the following conditions 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 these conditions.

- White space:

  Defined as a space, form-feed, newline, carriage return, horizontal tab, or vertical tab. The following Unicode characters are recognized as white space:

  - \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 (identifier 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 that is used to escape the special meaning of the _ and % in *pattern-value*. The LIKE operator must be used only to compare two string values.
  - `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.

  Note: The LIKE operator must be used to compare two string values. The value of `Root.MQMD.CorrelId` is a 24-byte byte array, not a character string. The selector string `Root.MQMD.CorrelId LIKE 'ABC%'` is accepted by the parser as syntactically valid, but it is evaluated to false. When you are comparing a byte array with a character string, LIKE therefore cannot be used.
- `identifier IS NULL` comparison operator tests for a NULL header field value, or a missing property value.
- `identifier IS NOT NULL` comparison operator tests for the existence of a non-null header field value or a property value.

- 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 TRUE and FALSE values.

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

Table 1. Value of the Boolean operator outcome when the 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. Value of the Boolean operator outcome when the 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. Value of the Boolean operator outcome when the logic is NOT A

Operator A	Outcome (NOT A)
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 > 2500"
```

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 alternative representation for an approximate numeric value.

SQL comments are not supported.

Related concepts

“Message properties” on page 26
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 IBM MQ and JMS applications.

Related reference

MsgHandle

MQBUFMH - Convert buffer into message handle

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.

- Message selection for publish/subscribe messaging occurs on the message as sent by the publisher. See Selection strings.
- Equivalence is tested using a single equals character; for example, `a = b` is correct, whereas `a == b` is incorrect.
- An operator used by many programming languages to represent 'not equal to' is `!=`. This representation is not a valid synonym for <> ; for example, `a <> b` is valid, whereas `a != b` is not valid.
- Single quotation marks are recognized only if the ' (U+0027) character is used. Similarly, double quotation marks, valid only when used to enclose byte strings, must use the " (U+0022) character.
- The symbols &, &&, | and || are not synonyms for logical conjunction/disjunction; for example, `a && b` must be specified 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. The parser evaluates operators that have the same precedence from left to right. The example would therefore 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 quotation marks; if single quotation marks are used, the byte string is taken to be a string literal. The number of characters (not the number that the characters represent) following the `0x` must be a multiple of two.
- The keyword `IS` is not a synonym for the equals character. 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` cases.

Related concepts

“Selection behavior” on page 33
Overview of IBM MQ selection behavior.

Related reference

Selection strings

UTF-8 and Unicode considerations when using message selectors

Characters, not enclosed in single quotation marks, that make up the reserved keywords of a selection string must be entered in Basic Latin Unicode (ranging from character U+0000 to U+0007F). It is not valid to use other code point representations of alphanumeric characters. For example, the number 1 must be expressed as U+0031 in Unicode, it is not valid to use the Fullwidth Digit equivalent U+FF11 or the Arabic equivalent U+0661.

Message property names can be specified using any valid sequence of Unicode characters. Message property names contained within selection strings that are encoded in UTF-8 will be validated even if they contain multi-byte characters. Validation of multi-byte UTF-8 is strict and you must ensure that valid UTF-8 sequences are used for message property names. Characters beyond the Unicode Basic Multilingual Plane (those above U+FFFF), represented in UTF-16 by surrogate code points (X'D800' through X'DFFF'), or four bytes in UTF-8, are not supported in message property names.

No extra processing is performed on property names or values when comparing for equality. This means for example that no pre/de-composition takes place and ligatures are not given any special meaning. For example, the pre-composed umlaut character U+00FC is not considered to be equivalent to U+0075 + U+0308 and the character sequence ff is not considered to be equivalent to the Unicode U+FB00 (LATIN SMALL LIGATURE FF)

Property data enclosed in single quotation marks can be represented by any sequence of bytes and is not validated.

Selecting on the content of a message

It is possible to subscribe based on a selection of message payload content (also known as content filtering), but the decision about which messages should be delivered to such a subscription cannot be performed directly by IBM MQ; instead an extended message selection provider, for example IBM Integration Bus, is required to process the messages.

When an application publishes on a topic string, where one or more subscribers have a selection string selecting on the content of the message, IBM MQ will request that the extended message selection provider parse the publication and inform IBM MQ whether the publication matches the selection criteria specified by each subscriber with a content filter.

If the extended message selection provider determines that the publication matches the subscriber's selection string, the message will continue to be delivered to the subscriber.

If the extended message selection provider determines that the publication does not match, the message is not delivered to the subscriber. This might cause the MQPUT or MQPUT1 call to fail with reason code MQRC_PUBLICATION_FAILURE. If the extended message selection provider is unable to parse the publication, reason code MQRC_CONTENT_ERROR is returned and the MQPUT or MQPUT1 call fails.

If the extended message selection provider is unavailable or is unable to determine whether the subscriber should receive the publication, reason code MQRC_SELECTION_NOT_AVAILABLE is returned and the MQPUT or MQPUT1 call fails.

When a subscription is being created with a content filter and the extended message selection provider is not available, the MQSUB call fails with reason code MQRC_SELECTION_NOT_AVAILABLE. If a subscription with a content filter is being resumed and the extended message selection provider is not available, the MQSUB call returns a warning of MQRC_SELECTION_NOT_AVAILABLE, but the subscription is allowed to be resumed.

Related reference

Selection strings

Asynchronous consumption of IBM MQ messages

Asynchronous consumption uses a set of Message Queue Interface (MQI) extensions, the MQI calls MQCB and MQCTL, which 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.

Callback
A generic term used to refer to either a Message Consumer or an Event Handler routine.

Asynchronous consumption can simplify the design and implementation of new applications, especially those that process multiple input queues or subscriptions. However, if you are using more than one input queue and you are processing messages in priority sequence, priority sequence is observed independently within each queue: you might get low-priority messages from one queue ahead of high-priority messages from another. Message order across multiple queues is not guaranteed. Also note that if you use API exits, you might need to change them to include the MQCB and MQCTL calls.

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

Figure 5 on page 41 shows a multithreaded application consuming messages from two queues. The example shows all of the messages being delivered to a single function.

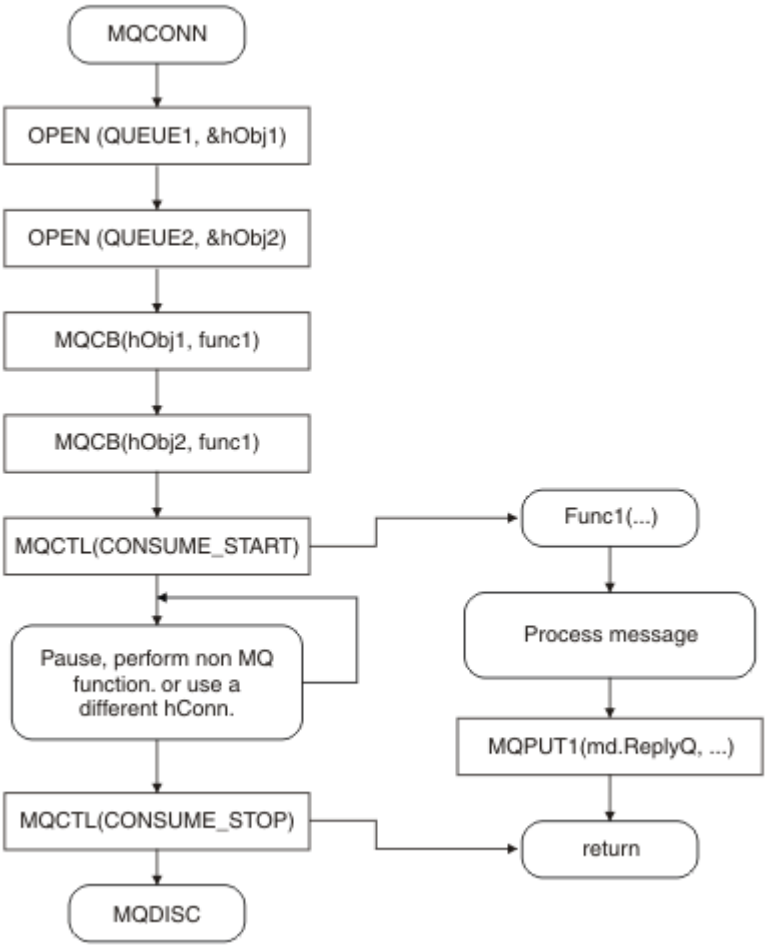


Figure 5. Standard Message Driven application consuming from two queues

z/OS On z/OS, the main control thread must issue an MQDISC call before ending. This allows any callback threads to end and release system resources.

Figure 6 on page 42 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 does not 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.

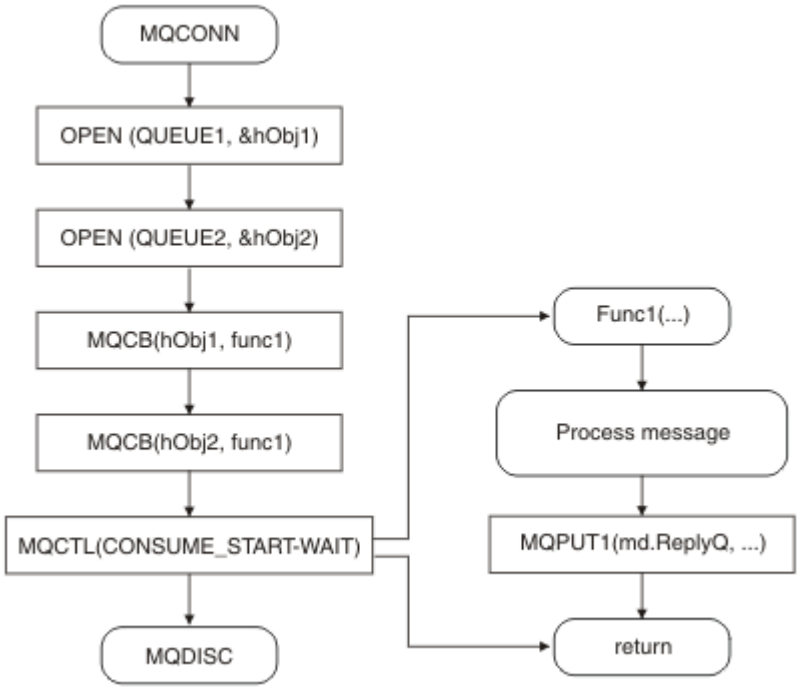


Figure 6. Single Threaded Message Driven application consuming from two queues

Message groups

Messages can occur within groups to allow ordering of messages.

Message groups allow multiple messages to be marked as related to one another, and a logical order to be applied to the group (see "Logical and physical ordering" on page 747). On Multiplatforms, message segmentation enables large messages to be broken up into smaller segments. You cannot use grouped or segmented messages when putting to a topic.

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 on page 42 shows a group of logical messages:

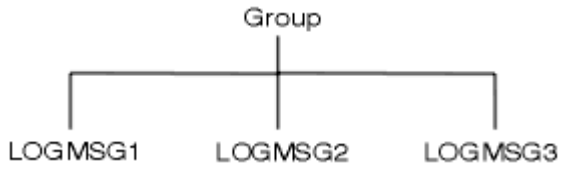


Figure 7. Group of logical messages

By opening a queue and specifying MQOO_BIND_ON_GROUP, you force all messages in a group that are sent to this queue to be sent to the same instance of the queue. For more information on the BIND_ON_GROUP option, see Handling message affinities.

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 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 to 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, and encodings.

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, see "Message segmentation" on page 764.

An individual message is broken down into smaller messages called *segments*. 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 on page 43 shows an example of messages within a group). A segment is logically part of a single message, so only the *MsgId*, *Offset*, and *MsgFlags* fields in the MQMD should differ between separate segments of the same message. If a segment fails to arrive, reason code MQRC_INCOMPLETE_GROUP or MQRC_INCOMPLETE_MSG is returned as appropriate.

Figure 8 on page 43 shows a group of logical messages, some of which are segmented:

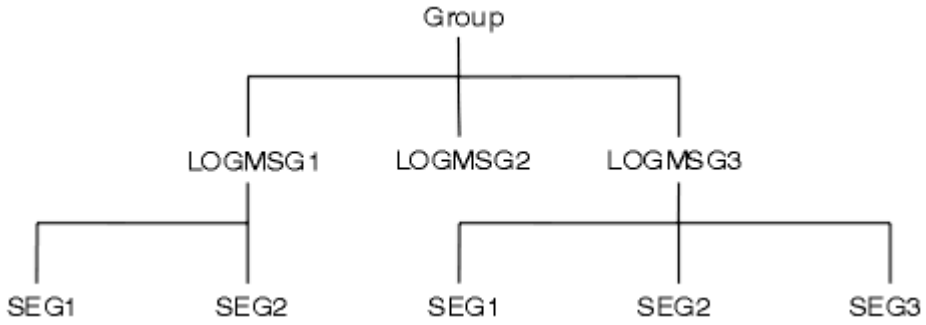


Figure 8. Segmented messages

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

You cannot use segmented or grouped messages with Publish/Subscribe.

Related concepts

"Message segmentation" on page 764
Use this information to learn about segmenting messages. This feature is not supported on IBM MQ for z/OS or by applications using IBM MQ classes for JMS.

Related reference

"Logical and physical ordering" on page 747
Within each priority level, messages on queues can occur in *physical* or *logical* order.

MQMD - Message descriptor

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.

z/OS Nonpersistent messages stored in a coupling facility (CF) on z/OS 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 sync point 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 823.
- z/OS ALW For all platforms except IBM i, a persistent message within the current unit of work is logged only when the unit of work is committed, and the unit of work can contain many queue operations.

Nonpersistent messages can be used for fast messaging. See Safety of messages for further information about fast messages.

Note: A combination of writing persistent messages within a unit of work, and writing persistent messages outside a unit or work, can cause potentially severe performance problems for your applications. This is particularly true when the same target queue is used for both operations.

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 procedural program errors" on page 999for 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.

Multi The backout count always survives restarts of the queue manager. Any change to the **HardenGetBackout** attribute is ignored.

z/OS For shared queues, the backout count always survives restarts of the queue manager. For all other configurations on 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 cost of extra processing.

For more information on committing and backing out messages, see "Committing and backing out units of work" on page 823.

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 1002).

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 about how to specify context information, see "Controlling message context information" on page 733.

The user context is used by the queue manager when generating the following types of report message:

- Confirm on delivery
- Expiry

When these report messages are generated, the user context is checked for +put and +passid authority on the destination of the report. Where the user context has insufficient authority, the report message is placed on the dead-letter queue if one has been defined. Where there is no dead-letter queue, the report message is discarded.

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. Suitably authorized applications can set the following fields:

- 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).

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

For information about how the queue manager fills the *UserIdentifier* and *AccountingToken* fields, see the descriptions of these fields in UserIdentifier and AccountingToken.

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* defines the type of application that put the message (for example, a CICS transaction).
- *PutApplName* defines the name of the application that put the message (for example, the name of a job or transaction).
- *PutDate* defines the date on which the message was put on the queue.
- *PutTime* defines the time at which the message was put on the queue.
- *ApplOriginData* defines 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 typically supplied by the queue manager. Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields. See the descriptions of these fields in PutDate and PutTime.

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.

IBM MQ objects

This information provides details on IBM MQ objects which include: queue managers, queue sharing groups, queues, administrative topic objects, namelists, process definitions, authentication information objects, channels, storage classes, listeners, and services.

Queue managers define the properties (known as attributes) of these objects. The values of these attributes affect the way in which IBM 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 IBM 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.

Related concepts

"Controlling message context information" on page 733
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. You can use the options field in the MQPMO structure to control context information.

Related reference

"MQOPEN options relating to message context" on page 724
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.

Windows Preparing and running Microsoft Transaction Server applications

To prepare an MTS application to run as an IBM MQ MQI client application, follow these instructions as appropriate for your environment.

For general information about how to develop Microsoft Transaction Server (MTS) applications that access IBM MQ resources, see the section on MTS in the IBM MQ Help Center.

To prepare an MTS application to run as an IBM MQ MQI client application, do one of the following for each component of the application:

- If the component uses the C language bindings for the MQI, follow the instructions in "Preparing C programs in Windows" on page 978 but link the component with the library mqicxa.lib instead of mqic.lib.
- If the component uses the IBM MQ C++ classes, follow the instructions in "Building C++ programs on Windows" on page 531 but link the component with the library imqx23vn.lib instead of imqc23vn.lib.
- If the component uses the Visual Basic language bindings for the MQI, follow the instructions in the "Preparing Visual Basic programs in Windows" on page 981 but when you define the Visual Basic project, type `MqType=3` in the **Conditional Compilation Arguments** field.

Design considerations for IBM MQ applications

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 IBM MQ.

When designing an IBM MQ application consider the following questions and options:

Type of application

What is the purpose of your application? See the following links for information about that different types of application you can develop:

- Server
- Client
- Publish/subscribe
- Web services
- User exits, API exits, and installable services

Additionally, you can also write your own applications to automate administration of IBM MQ. For more information, see The IBM MQ Administration Interface (MQAI) and Automating administration tasks.

Programming language

IBM MQ supports a number of different programming languages for writing applications. For more information see, "Developing applications for IBM MQ" on page 5.

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.

For example 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
```

Types of queues

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.

z/OS **Using shared queues, queue sharing groups, and queue sharing group clusters (IBM 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 Shared queues and queue sharing groups for more information.

You might also want to estimate the average and peak message flows and consider using queue sharing group clusters to spread the workload. See Shared queues and queue sharing groups for more information.

Using 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.

Types of messages

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. For more information about designing messages, see “Design techniques for messages” on page 56.

Using publish/subscribe or point-to-point messaging

Using publish/subscribe messaging, a sending application sends the information that it wants to share in an IBM MQ message to a standard destination managed by IBM MQ publish?subscribe, and lets IBM 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 Publish/subscribe messaging.

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.

Controlling your IBM MQ programs

You might want to start some programs automatically or make programs wait until a particular message arrives on a queue (using the IBM MQ *triggering* feature, see “Starting IBM MQ applications using triggers” on page 834). Alternatively, you might want to start another instance of an application when the messages on a queue are not getting processed fast enough (using the IBM MQ *instrumentation events* feature as described in Instrumentation events).

Running your application on an IBM MQ client

The full MQI is supported in the client environment, and almost any IBM MQ application written in a procedural language can be relinked to run on an IBM MQ MQI client. Link the application on the IBM MQ MQI client to the MQIC library, rather than to the MQI library. z/OS Get(signal) on z/OS is not supported.

Note: An application running on an IBM 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 "Running applications in the IBM MQ MQI client environment" on page 887 for more information.

Application performance

Design decisions can impact your application performance, for suggestions for enhancing performance of IBM MQ applications, see "Application design and performance considerations" on page 57 IBM i and "Design and performance considerations for IBM i applications" on page 61 .

Advanced IBM MQ techniques

For more advanced applications you might want to use some advanced IBM MQ techniques such as correlating replies, and generating and sending IBM MQ context information. For more information, see "Design techniques for advanced applications" on page 59.

Securing your data and maintaining 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 IBM 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 823 for further details). You can use the *persistence* feature of IBM MQ messages to assure the delivery of important messages.

Testing IBM MQ applications

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

z/OS When testing CICS applications with IBM 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 about how to do this, see "Using and writing applications on IBM MQ for z/OS" on page 857.

IBM i When testing IBM i applications, you can use the standard Debugger. To start this, use the STRDBG command.

Handling 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.

Related concepts

IBM MQ technical overview

"Design and performance considerations for z/OS applications" on page 62
Application design is one of the most important factors affecting performance. Use this topic to understand some of the design factors involved in performance.

"Developing applications for IBM MQ" on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

"Application development concepts" on page 6

You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

"Writing a procedural application for queuing" on page 696
Use this information to learn about writing queuing applications, connecting to and disconnecting from a queue manager, publish/subscribe, and opening and closing objects.

"Writing client procedural applications" on page 879
What you need to know to write client applications on IBM MQ using a procedural language.

"Developing C++ applications" on page 508
IBM MQ provides C++ classes equivalent to IBM MQ objects and some additional classes equivalent to the array data types. It provides a number of features not available through the MQI.

"Using IBM MQ classes for JMS/Jakarta Messaging" on page 79
IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

"Using IBM MQ classes for Java" on page 335
Use IBM MQ in a Java environment. IBM MQ classes for Java allow a Java application to connect to IBM MQ as an IBM MQ client, or connect directly to an IBM MQ queue manager.

Related tasks

"Developing .NET applications" on page 535
IBM MQ classes for .NET allow .NET applications to connect to IBM MQ as an IBM MQ MQI client or to connect directly to an IBM MQ server.

Specifying the application name in supported programming languages

Before IBM MQ 9.2.0, you could already specify an application name on Java or JMS client applications. From IBM MQ 9.2.0 this feature is extended to other programming languages on IBM MQ for Multiplatforms.

How the application name is used

The application name is output from:

- runmqsc DISPLAY CONN APPLTAG
- runmqsc DISPLAY QSTATUS TYPE(HANDLE) APPLTAG
- runmqsc DISPLAY CHSTATUS RAPPLTAG
- MQMD.PutApplName
- Application activity trace

The application name is also used when configuring application activity trace. The default application name for non-Java applications is the truncated name of the executable, except on Windows and IBM i.

Windows On Windows, the default name is the fully qualified executable name, truncated to 28 characters on the left.

IBM i On IBM i, the default name is the job name.

For Java applications it is the class name prefixed by the package name truncated on the left to 28 characters.

For more information, see PutApplName.

Applications on IBM MQ for Multiplatforms can set their application names either administratively or by using various programming methods. This enables applications to provide a more meaningful platform-independent name, when you configure application activity trace or when output from various **runmqsc** commands.

You can rebalance applications across a uniform cluster. Meaningful application names are used to achieve this.

Supported characters

See “Recommended application name characters” on page 51 for more information on how you specify the application name.

Programming languages

See “Programming language connections” on page 53 for more information on how applications resolving to the IBM MQ libraries in C, and other programming languages, can provide the application name.

Managed .NET applications

See “Managed .NET applications” on page 54 for information on how managed .NET applications can provide the application name.

XMS applications

See “XMS applications” on page 55 for information on how XMS applications can provide the application name.

Java and JMS bindings applications

ALW

See “Java and JMS bindings applications” on page 55 for information on how Java and JMS applications can provide the application name.

Related concepts

Application activity trace

About uniform clusters

Related reference

MQCNO

MQCNO on IBM i

Using the application name in supported programming languages

Use this information to learn how the application name is selected in the various languages that IBM MQ supports.

Recommended application name characters

Application names must be in the character set given by the **CodedCharSetId** attribute of the queue manager field. For more information on this attribute, see Attributes for the queue manager.

However, if the application is running as an IBM MQ MQI client, the application name must be in the character set and encoding of the client.

To ensure a smooth transition of the application name between queue managers, and to allow application resource monitoring through the resource monitoring topics, application names should contain only single-byte printable characters.

Notes:

- You should also avoid the use of forward slash and ampersand characters in application names.

- You should avoid use of the ampersand character in application names. System topic STATAPP metrics for application names containing an ampersand will not be produced.

This limits the name to:

- Alphanumeric characters: `A-Z`, `a-z`, and `0-9`

  Note: You should not use the lowercase `a-z` characters in application names on systems using EBCDIC Katakana.
- The space character
- Printable characters that are invariant in EBCDIC: `+ < = > % * ' ( ) , _ - . : ; ?`
- The / character. When subscribing to activity trace or STATAPP system topic metrics for an application whose name contains a forward slash, you must replace any forward slash characters with an ampersand character. For example, to receive STATAPP metrics for an application called "DEPT1/APPS/STOCKQUOTE" you must subscribe to the topic string "$SYS/MQ/INFO/QMGR/QMBASIC/Monitor/STATAPP/DEPT1&APPS&STOCKQUOTE/INSTANCE". The amqsact and amqsrua sample applications will automatically convert forward slash characters to ampersands when creating their subscriptions.

How you set the characters

The following table summarizes the means by which the application name is chosen in the various languages IBM MQ supports. The means by which the name is chosen is in order of precedence, highest first.

	C bindings and client	**Java bindings and client**	**JMS bindings and client**	**Managed .NET client**	**Unmanaged .NET bindings and client**	**Managed XMS client**	**Unmanaged .XMS bindings and client**
Connection property override		Java connection property override		.NET connection property override	.NET connection property override		
Overridden property		Java overridden property		.NET overridden property	.NET overridden property		
MQEnvironment		Java MQEnvironment		.NET MQEnvironment	.NET MQEnvironment		
Connection factory property			Connection factory property			Connection factory property	Connection factory property
JMSAdmin			JMSAdmin			JMSAdmin	JMSAdmin

	C bindings and client	Java bindings and client	JMS bindings and client	Managed .NET client	Unmanaged .NET bindings and client	Managed XMS client	Unmanaged .XMS bindings and client
MQCNO	Connection options						
Environment variable	Environment variables				Environment variables		Environment variables
mqclient.ini (Applicable to client connections only)	Client connections				Client connections		Client connections
Java class name		Java class name	Java class name				
Default name	Default name			.NET Default name	.NET Default name	.NET Default name	.NET Default name

Note: The C bindings and client column applies to the following programming languages as well:

- COBOL
- Assembler
- Visual Basic
- RPG

Programming language connections

Applications resolving to the IBM MQ libraries in C, and other programming languages, can provide the application name in the following ways.

The methods of connection are listed in order of precedence, starting with the highest.

Multi **Connection options**

- ALW MQCNO

  Note: z/OS When connecting to an IBM MQ for z/OS queue manager, you can only set the application name by using client mode connections, or by using IBM MQ classes for JMS or IBM MQ classes for Java applications.

- IBM i MQCNO on IBM i

ALW **Environment variables**

If you have not yet selected an application name, you can use the **MQAPPLNAME** environment variable to identify the connection to the queue manager. For example:

```
export MQAPPLNAME=ExampleApplName
```

Note that the first 28 characters only are used, and these characters must not be all blanks or nulls.

Note: The attribute applies to the supported programming languages, unmanaged .NET, and unmanaged XMS connections only.

ALW **Client configuration file**

If the you have not yet selected an application name, and the connection is a client connection, you can specify the following information in the client configuration file (for example, `mqclient.ini`) to identify the connection to the queue manager.

```
Connection:
    ApplName=ExampleApplName
```

Notes:

1. The first 28 characters only are used, and these characters must not be all blanks or nulls.
2. The attribute applies only to client connections on the supported programming languages, unmanaged .NET, and unmanaged XMS connections only.

For more information, see IBM MQ MQI client configuration file, `mqclient.ini`.

Default name

If you have still not chosen the application name, the default name continues to be used, which contains as much of the path and executable name as the operating system displays. For more information, see PutApplName.

Managed .NET applications

Managed .NET applications can provide the application name in the following ways.

The methods of connection are listed in order of precedence, starting with the highest.

Connection property override

You can provide a connection details override file to applications in the following way:

```
<appSettings>
     <add key="overrideConnectionDetails" value="true" />
     <add key="overrideConnectionDetailsFile" value="<location>" />
</appSettings>
```

The file specified by `overrideConnectionDetailsFile` contains a list of properties prefixed by `mqj`. Applications need to define the **`mqj.APPNAME`** property where the value of the **`mqj.APPNAME`** property specifies the name used to identify the connection to the queue manager.

Only the first 28 characters of the name are used. For example:

```
mqj.APPNAME=ExampleApplName
```

Overridden property

A constant **`MQC.APPNAME_PROPERTY`** has been defined with the value *APPNAME*. You can now pass this property to the **`MQQueueManager`** constructor, using the first 28 characters only of the name. For example:

```
Hashtable properties = new Hashtable();
properties.Add( MQC.APPNAME_PROPERTY, "ExampleApplName" );
MQQueueManager qMgr = new MQQueueManager("qmgrname", properties);
```

For more information, see “Managed and unmanaged operations in .NET” on page 612.

MQEnvironment

The *AppName* property is added to the **MQEnvironment**class, and the first 28 characters only are used. For example:

```
MQEnvironment.AppName = "ExampleApplName";
```

Default name

If you have not provided the application name by any of the means described in the preceding text, the application name is automatically set to be the executable name (and as much of the path that will fit).

XMS applications

The methods of connection are listed in order of precedence, starting with the highest.

Connection factory property

XMS applications can provide the application name on the connection factory using the **XMSC.WMQ_APPLICATIONNAME** property ("*XMSC_WMQ_APPNAME*") in a similar way to JMS. You can specify up to 28 characters.

For more information, see "XMS .NET creating administered objects" on page 639 and "Properties of an XMS message" on page 646.

JMSAdmin

In the administrative tooling the property is known as "**APPLICATIONNAME**" or "**APPNAME**" for short.

Java and JMS bindings applications

The methods of connection are listed in order of precedence, starting with the highest.

ALW Java and JMS client applications can already specify an application name, and this has been extended on IBM MQ for Multiplatforms to bindings applications, by making use of the MQCNO **ApplName** field.

Connection property override

The **Application name** property has been added to the list of connection properties that you can override. For more information, see Using IBM MQ connection property override.

Attention: The connection properties and the way to use the Connection Property Override file is the same for both IBM MQ classes for Java and .NET.

Overridden property

A constant **MQC.APPNAME_PROPERTY** has been defined with the value *APPNAME*. You can now pass this property to the **MQQueueManager** constructor, using the first 28 characters only of the name. For more information, see Using connection property override in IBM MQ classes for Java.

MQEnvironment

The *AppName* property is added to the **MQEnvironment** class, and the first 28 characters only are used.

For more information, see "Setting up the IBM MQ environment for IBM MQ classes for Java" on page 361.

Java class name

If you have not provided the application name by any of the means in the preceding text, the application name is derived from the main class name.

For more information, see “Setting up the IBM MQ environment for IBM MQ classes for Java” on page 361.

Attention: IBM i On IBM i it is not possible to query the main class name, so IBM MQ client for Java is used instead.

Related concepts

“Setting up the IBM MQ environment for IBM MQ classes for Java” on page 361
For an application to connect to a queue manager in client mode, the application must specify the channel name, host name, and port number.

Related reference

MQCNO

MQCNO on IBM i

Design techniques for messages

Considerations to help you design messages, including considerations for selectors and message properties.

Things to consider at the design stage

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, because they affect the way that you create your messages:

Type of message to 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.

Assigning different priorities to messages

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.

Effect of restarting queue manager on messages

The queue manager preserves all persistent messages, recovering them when necessary from the IBM 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 IBM MQ for Windows or IBM MQ on AIX and Linux 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.

z/OS Because messages on shared queues (only available on IBM 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.

Giving information about yourself to the recipient of 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.

Amount of receiving queues

Multi If a message might need to be put on several queues, you can publish to a topic or a distribution list.

z/OS If a message might need to be put on several queues, you can publish to a topic.

Selectors and message properties

Messages can have metadata associated with them alongside the main message payload. These message properties can be useful in supplying additional data.

There are two aspects to this additional data that it is important to know about:

- The properties are not subject to Advanced Message Security (AMS) protection. If you want to use AMS to protect your data, then put it in the payload and not the message properties.
- The properties can be used to perform the selection of messages.

It is important to note that using selectors breaks the standard message convention of first in first out. As the queue manager is optimized for this workload, providing complex selectors is not advised for performance reasons. The queue manager does not store indexes of the message properties, therefore searching for a message must be a linear search. The deeper the queue, the more complex the selector, and the lower probability that the selector matching a message will adversely affect performance.

If complex selection is required, it is suggested to filter the messages by using any application or processing engine, such as IBM Integration Bus, to different destinations. Alternatively, the use of a topic hierarchy might be useful.

Note: IBM MQ classes for Java do not support the use of selectors, if you do wish to use selectors these should be done via the JMS API.

Application design and performance considerations

There are a number of ways in which poor program design can affect performance. These can be difficult to detect because the program can appear to perform well itself, but affect the performance of other tasks. Several problems specific to programs making IBM MQ calls are explained in this topic.

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.

Effect of message length

The amount of data in a message can affect the performance of the application that processes the message. To achieve the best performance from your application, send only the essential data in a message. For example, in a request to debit a bank account, the only information that might need to be passed from the client to the server application is the account number and the amount of the debit.

Effect of message persistence

Persistent messages are usually logged. Logging messages reduces the performance of your application, so use persistent messages for essential data only. If the data in a message can be discarded if the queue manager stops or fails, use a nonpersistent message.

z/OS MQPUT and MQGET operations for persistent messages will block when there is insufficient recovery log space to record the operations. Such a condition is indicated in the queue manager job log by messages CSQJ110E and CSQJ111A. Ensure monitoring processes are in place so that such conditions are managed and avoided.

Searching for a particular message

The MQGET call usually retrieves the first message from a queue. If you use the message and correlation identifiers (*MsgId* and *CorrelId*) in the message descriptor to specify a particular message, the queue manager has to search the queue until it finds that message. Using the MQGET call in this way affects the performance of your application.

Queues that contain messages of different lengths

If your application cannot use messages of a fixed length, grow and shrink the buffers dynamically to suit the typical message size. If the application issues an MQGET call that fails because the buffer is too small, the size of the message data is returned. Add code to your application so that the buffer is resized accordingly and the MQGET call is reissued.

Note: If you do not set the **MaxMsgLength** attribute explicitly, it defaults to 4 MB, which might be very inefficient if this is used to influence the application buffer size.

Frequency of sync points

Programs that issue very large numbers of MQPUT or MQGET calls within sync point, without committing them, can cause performance problems. Affected queues can fill up with messages that are currently inaccessible, while other tasks might be waiting to get these messages. This has implications in terms of storage, and in terms of threads that are tied up with tasks that are attempting to get messages.

Use of the MQPUT1 call

Use the MQPUT1 call only if you have a single message to put on a queue. If you want to put more than one message, use the MQOPEN call, followed by a series of MQPUT calls and a single MQCLOSE call.

Number of threads in use

Windows For IBM MQ for Windows, an application might require a large number of threads. Each queue manager process is allocated a maximum allowable number of application threads.

Applications might use too many threads. Consider whether the application takes into account this possibility and that it takes actions either to stop or to report this type of occurrence.

Put persistent messages under syncpoint

Persistent messages should be put and got under syncpoint. This is because when getting a persistent message outside of syncpoint, if the get fails, there is no way for the application to know whether the message has been got from the queue or not, and whether, if the message has been got, then it has also been lost. When getting persistent messages under syncpoint, if anything fails, the transaction is rolled back and the persistent message is not lost because it is still on the queue.

Similarly, when putting persistent messages, put them under syncpoint. Another reason for putting and getting persistent messages under syncpoint is that the persistent message code in IBM MQ is heavily optimized for syncpoint. So putting and getting persistent messages under syncpoint is faster than putting and getting persistent messages outside of syncpoint.

If your application does put persistent messages outside syncpoint, the queue manager checks to see if it can create an implicit syncpoint on behalf of the application. If the queue manager can do so, it includes the put inside that syncpoint, and commits it automatically. See “Implicit syncpoint on Multiplatforms” on page 831 for a more detailed description.

However, it is faster to put and get non-persistent messages outside of syncpoint because the nonpersistent code in IBM MQ is optimized for being outside of syncpoint. Putting and getting persistent messages go at disk speeds because the persistent message is persisted to disk. However, putting and getting non-persistent messages go at CPU speeds because there is no disk write involved, not even when using syncpoint.

If an application is getting messages and does not know in advance whether they are persistent or not, the GMO option MQGMO_SYNCPOINT_IF_PERSISTENT can be used.

Design techniques for advanced applications

When designing more advanced applications, there are some techniques that you might want to consider such as waiting for messages, correlating replies, setting and using context information, starting applications automatically, generating reports and removing message affinities when using clustering.

For a simple IBM MQ application, you need to decide which IBM 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 769).
- z/OS On IBM MQ for z/OS only, setting a signal so that the program is informed when a message arrives. For more information, see “Signaling” on page 770.
- Establishing a callback exit to be driven when a message arrives; see “Asynchronous consumption of IBM MQ messages” on page 40.
- Making periodic calls on the queue to see whether a message has arrived (*polling*). This is not typically advisable because it can have performance implications.

Correlating replies

In IBM MQ applications, when a program receives a message that requests it to do some work, the program typically 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 about using and setting context information, see "Message context" on page 45.

Starting IBM MQ programs automatically

Use IBM 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 about triggering, see "Starting IBM MQ applications using triggers" on page 834. Triggering is just one way of starting a program automatically. For example, you can start a program automatically on a timer using non-IBM MQ facilities.

Multi On Multiplatforms, IBM MQ can define service objects to start IBM MQ programs when the queue manager starts; see Service objects.

Generating IBM 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 Handling message affinities.

Design and performance considerations for IBM i applications

Use this information to understand how application design, threads, and storage, can affect performance.

This information is split into two sections:

- "Application design considerations" on page 61
- "Specific performance problems" on page 62

Application design considerations

There are a number of ways in which poor program design can affect performance. These problems can be difficult to detect because the program can appear to perform well, while affecting the performance of other tasks. Several problems specific to programs making IBM MQ for IBM i calls are explained in the following sections.

For more information about application design, see "Design considerations for IBM MQ applications" on page 47.

Effect of message length
Although IBM MQ for IBM i allows messages to hold up to 100 MB of data, the amount of data in a message affects the performance of the application that processes the message. To achieve the best performance from your application, send only the essential data in a message; for example, in a request to debit a bank account, the only information that might need to be passed from the client to the server application is the account number and the amount of the debit.

Effect of message persistence
Persistent messages are journaled. Journaling messages reduces the performance of your application, so use persistent messages for essential data only. If the data in a message can be discarded if the queue manager stops or fails, use a nonpersistent message.

Searching for a particular message
The MQGET call usually retrieves the first message from a queue. If you use the message and correlation identifiers (*MsgId* and *CorrelId*) in the message descriptor to specify a particular message, the queue manager must search the queue until it finds that message. The use of the MQGET call in this way affects the performance of your application.

Queues that contain messages of different lengths
If the messages on a queue are of different lengths, to determine the size of a message, your application can use the MQGET call with the *BufferLength* field set to zero so that, even though the call fails, it returns the size of the message data. The application can then repeat the call, specifying the identifier of the message it measured in its first call and a buffer of the correct size. However, if there are other applications serving the same queue, you might find that the performance of your application is reduced because its second MQGET call spends time searching for a message that another application has retrieved in the time between your two calls.

If your application cannot use messages of a fixed length, another solution to this problem is to use the MQINQ call to find the maximum size of messages that the queue can accept, then use this value in your MQGET call. The maximum size of messages for a queue is stored in the **MaxMsgLen** attribute of the queue. This method might use large amounts of storage, however, because the value of this queue attribute can be the maximum allowed by IBM MQ for IBM i, which might be greater than 2 GB.

Frequency of sync points
Programs that issue numerous MQPUT calls within sync point, without committing them, can cause performance problems. Affected queues can fill up with messages that are currently unusable, while other tasks might be waiting to get these messages. This problem has implications in terms of storage, and in terms of threads tied up with tasks that are attempting to get messages.

Use of the MQPUT1 call
Use the MQPUT1 call only if you have a single message to put on a queue. If you want to put more than one message, use the MQOPEN call, followed by a series of MQPUT calls and a single MQCLOSE call.

Number of threads in use

An application might require many threads. Each queue manager process is allocated a maximum allowable number of threads. If some applications are troublesome, it might be due to their design using too many threads. Consider whether the application takes into account this possibility and that it takes actions either to stop or to report this type of occurrence. The maximum number of threads that IBM i allows is 4,095. However, the default is 64. IBM MQ makes available up to 63 threads to its processes.

Specific performance problems

This section explains the problems of storage and poor performance.

Storage problems

If you receive the system message `CPF0907. Serious storage condition may exist` it is possible that you are filling up the space associated with the IBM MQ for IBM i queue managers.

Is your application or IBM MQ for IBM i running slowly?

If your application is running slowly, it might indicate that it is in a loop, or waiting for a resource that is not available. This slow running might also be caused by a performance problem. Perhaps it is because your system is operating near the limits of its capacity. This type of problem is probably worst at peak system load times, typically at mid-morning and mid-afternoon. (If your network extends across more than one time zone, peak system load might seem to you to occur at some other time.)

If you find that performance degradation is not dependent on system loading, but happens sometimes when the system is lightly loaded, a poorly designed application program is probably to blame. This problem might manifest itself as a problem that only occurs when certain queues are accessed.

QTOTJOB and QADLTOTJ are system values worth investigating.

The following symptoms might indicate that IBM MQ for IBM i is running slowly:

- If your system is slow to respond to MQSC commands.
- If repeated displays of the queue depth indicate that the queue is being processed slowly for an application with which you would expect a large amount of queue activity.
- Is IBM MQ trace running?

Linux Design considerations for Linux on Power Systems - Little Endian applications

As Linux on Power® Systems - Little Endian supports 64-bit applications only, there is no support provided in IBM MQ for 32-bit applications.

Related concepts

“Design considerations for IBM MQ applications” on page 47

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 IBM MQ.

z/OS Design and performance considerations for z/OS applications

Application design is one of the most important factors affecting performance. Use this topic to understand some of the design factors involved in performance.

There are a number of ways in which poor program design can affect performance. These problems can be difficult to detect because the program can appear to perform well, while affecting the performance of other tasks. Several problems specific to programs making MQI calls are demonstrated in the following sections.

For more information about application design, see “Design considerations for IBM MQ applications” on page 47.

Effect of message length

Although IBM MQ for z/OS allows messages to hold up to 100 MB of data, the amount of data in a message affects the performance of the application that processes the message. To achieve the best performance from your application, send only the essential data in a message. For example, in a request to debit a bank account, the only information that might need to be passed from the client to the server application is the account number and the amount to debit.

Effect of message persistence

Persistent messages are logged. Logging messages reduces the performance of your application, so use persistent messages for essential data only. If the data in a message can be discarded if the queue manager stops or fails, use a nonpersistent message.

Data for persistent messages is written to log buffers. These buffers are written to the log data sets when:

- A commit occurs
- A message is got or put out of syncpoint
- WRTHRSH buffers are filled

Processing many messages in one unit of work can cause less input/output than if the messages were processed one for each unit of work, or out of syncpoint.

Searching for a particular message

The MQGET call typically retrieves the first message from a queue. If you use the message and correlation identifiers (**MsgId** and **CorrelId**) in the message descriptor to specify a particular message, the queue manager searches the queue until it finds that message. Using MQGET in this way affects the performance of your application because, to find a particular message, IBM MQ might have to scan the entire queue.

You can use the **IndexType** queue attribute to specify that you want the queue manager to maintain an index that can be used to increase the speed of MQGET operations on the queue. However, there is a small performance reduction for maintaining an index, so only generate one if you need to use it. You can choose to build an index of message identifiers or of correlation identifiers, or you can choose not to build an index for queues where messages are retrieved sequentially. Try to have many different key values, not lots with the same value. For example Balance1, Balance2, and Balance3, not three with Balance. For shared queues, you must have the correct **IndexType**. For details of the **IndexType** queue attribute, see IndexType.

To avoid affecting queue manager restart time by using indexed queues, use the QINDXBLD(NOWAIT) parameter in the CSQ6SYSP macro. This allows the queue manager restart to complete without waiting for queue index building to complete.

For a full description of the **IndexType** attribute, and other object attributes see Attributes of objects.

Queues that contain messages of different lengths

Get a message, using a buffer size matching the expected size of the message. If you receive the return code indicating that the message is too long, get a bigger buffer. When the get fails in this way, the data length returned is the size of the unconverted message data. If you specify MQGMO_CONVERT on the MQGET call, and the data expands during conversion, it still might not fit in the buffer, in which case you need to further increase the size of the buffer.

If you issue the MQGET with a buffer length of zero, it returns the size of the message and the application can then get a buffer of this size and reissue the get. If you have multiple applications processing the queue, another application might have already processed the message when the original application reissued the get. If you occasionally have large messages, you might need to get a large buffer just for

these messages, and release it after the message has been processed. This should help reduce virtual storage problems if all applications have large buffers.

If your application cannot use messages of a fixed length, another solution to this problem is to use the MQINQ call to find the maximum size of messages that the queue can accept, then use this value in your MQGET call. The maximum size of messages for a queue is stored in the **MaxMsgL** attribute of the queue. This method could use large amounts of storage, however, because the value of **MaxMsgL** could be as high as 100 MB, the maximum allowed by IBM MQ for z/OS.

Note: You can lower the **MaxMsgL** parameter after large messages have been put to the queue. For example you can put a 100 MB message, then set **MaxMsgL** to 50 bytes. This means that it is still possible to get bigger messages than the application expected.

Frequency of syncpoints

Programs that issue many MQPUT calls within syncpoint, without committing them, can cause performance problems. Affected queues can fill up with messages that are currently unusable, while other tasks might be waiting to get these messages. This has implications in terms of storage, and in terms of threads tied up with tasks that are attempting to get messages.

As a rule if you have multiple applications processing a queue you typically get the best performance when you have either

- 100 short messages (less than 1 KB), or
- One message for larger messages (100 KB)

for each syncpoint. If there is only one application processing the queue, you must have more messages for each unit of work.

You can limit the number of messages that a task can get or put within a single unit of recovery with the **MAXUMSGS** queue manager attribute. For information about this attribute, see the **ALTER QMGR** command in MQSC commands.

Advantages of the MQPUT1 call

Use the MQPUT1 call only if you have a single message to put on a queue. If you want to put more than one message, use the MQOPEN call, followed by a series of MQPUT calls and a single MQCLOSE call.

How many messages can a queue manager contain

Local Queues

The number of local messages a queue manager can hold is basically the size of the page sets. You can have up to 100 page sets (though it is recommended page set 0 and page set 1 are for system related objects and queues). You can use a page set with extended format and increase the capacity of a page set.

Shared Queues

The capacity for shared queues depends on the size of the coupling facility (CF). IBM MQ uses CF list structures where fundamental storage units are entries and elements. Each message is stored as 1 entry and multiple elements containing the associated MQMD and other message data. The number of elements consumed by a single message depends on the size of the message and, for CFLEVEL(5), the offload rules in effect at MQPUT time. Fewer elements are needed when message data is offloaded to either Db2 or SMDS. Message data access is slower when the message has been offloaded. See Performance Supportpac MP1H for further comparison of performance and CPU overhead associated with message offload.

What affects performance

Performance can mean how fast messages can be processed, and it can also mean how much CPU is needed per message.

What affects how fast messages can be processed

For persistent messages the biggest impact is the speed of the log data sets. The speed of the log data sets depends on the DASD they are on. Therefore care should be taken to put log data set on low used volumes to reduce contention. Striping the MQ logs improves the log performance when there are multiple pages written per I/O. Z High Performance Fibre connection (zHPF) also has a significant performance to I/O response time when the I/O subsystem is busy.

When there is a request to get and put a message, access to the queue is locked during the request to preserve integrity of the queue. For planning purposes consider the queue locked for the whole request. So if the time for a put is 100 microseconds, and you have more than 10,000 requests a second you might experience delays. You might achieve better than this in practice, but it is a good general rule. You can use different queues to improve performance.

Possible reasons for this can be:

- use a common reply queue which every CICS transaction uses
- each CICS transaction is given a unique reply to queue
- a reply to a queue for CICS region and all transactions in the CICS region use this queue.

The answer depends on the number of requests a second, and the response time of the requests.

If messages have to be read from a page set, they will be slower compared to when the messages are in the buffer pool. If you have more messages than fit into a buffer pool, then they will spill to disk. So you need to ensure that the buffer pool is big enough for your short lived messages. If you have messages that you process many hours later, these are likely to spill to disk, so you should expect a get for these messages to be slower than if they were in the buffer pool.

For a shared queue, the speed of the messages depends on the speed of the Coupling Facility. A CF within the physical processor is likely to be faster than an external CF. The CF response time depends on how busy the CF is. For example on the Hursley systems, when the CF was 17% busy the response time was 14 microseconds. When the CF was 95% busy the response time was 45 microseconds.

If your MQ requests use a lot of CPU, this can affect how fast messages are processed. Because if the Logical Partition (LPAR) is constrained for CPU, applications will be delayed waiting for CPU.

How much CPU per message

In general bigger messages use more CPU, so avoid large (x MB) messages if possible.

When getting specific messages from queues, the queue should be indexed so the queue manager can go directly to the message (and so avoids potentially an entire scan of the queue). If the queue is not indexed then the queue is scanned from the beginning looking for the message. If there are 1000 messages on the queue, it may have to scan all 1000 messages. The result is a lot of unnecessary CPU usage.

Channels using TLS have an additional cost due to the encryption of the message.

In MQ V7 you can select messages by a selector string in addition to the **CORRELID** or **MSGID**. Every message has to be looked in, so if there are many messages on the queue this is expensive.

It is more efficient for an application to do OPEN PUT PUT CLOSE than PUT1 PUT1.

Triggering in CICS

When the message arrival rate of messages for a triggered queue is low, it is efficient to use trigger first. When the message arrival rate is more than 10 messages a second, it is more efficient to trigger the first transaction, then have the transaction process a message and get the next message, and so on. If a message has not arrived in a short period (say between 0.1 and 1 second) the transaction ends. At high throughput you might need multiple transactions running to process the messages and

to prevent a build up of messages. For every trigger message produced, this requires a put and a get of a trigger message, which in effect doubles the cost of the message.

How many connections or concurrent users are supported

Each connection uses virtual storage within the queue manager, so the more concurrent users the more storage used. If you need a very large buffer pool and large number of users, then you might be constrained for virtual storage, and you might need to reduce the size of your buffer pools.

If security is being used, the queue manager caches information within the queue manager for a long period. The amount of virtual storage that is used within the queue manager is affected.

The **CHINIT** can support up to about 10,000 connections. This is limited by virtual storage. If a connection uses more storage, for example using by TLS, the storage per connection increases, which therefore means the **CHINIT** can support less connections. If you are processing large messages, these will require more storage for buffers in the **CHINIT**, so the **CHINIT** can support less messages.

Connections to a remote queue manager are more efficient than client connections. For example, every MQ client requests requires two network flows (one for the request and one for the response). With a channel to a remote queue manager, there may be 50 sends over the network before a response comes back. If you are considering a large client network, it may be more efficient to use a concentrator queue manager on a distributed box, and have one channel coming in and out of the concentrator.

Other things affecting performance

Log data set should be at least 1000 cylinders in size. If the logs are smaller than this, checkpoint activity may be too frequent. On a busy system a checkpoint typically should be every 15 minutes or longer, at very high throughputs it may less than this. When a checkpoint occurs the buffer pools are scanned and 'old' messages and changed pages are written to disk. If checkpoints are too frequent, this can impact performance. The value of LOGLOAD can also affect checkpoint frequency. If the queue manager abnormally ends, then at restart it may have to read back to 3 checkpoints. The best checkpoint interval is a balance between the activity when a checkpoint is taken, and the amount of log data that may need to be read when the queue manager restarts.

There is a significant overhead incurred when starting a channel. It is usually better to start a channel and leave it connected, rather than frequent starts and stops of the channel.

Related information

MP1K: IBM MQ for z/OS 9.0 Performance Report

z/OS IMS and IMS bridge applications on IBM MQ for z/OS

This information helps you to write IMS applications using IBM MQ.

- To use syncpoints and MQI calls in IMS applications, see “Writing IMS applications using IBM MQ” on page 67.
- To write applications that use the IBM MQ - IMS bridge, see “Writing IMS bridge applications” on page 71.

Use the following links to find out more about IMS and IMS bridge applications on IBM MQ for z/OS:

- “Writing IMS applications using IBM MQ” on page 67
- “Writing IMS bridge applications” on page 71

Related concepts

“The Message Queue Interface overview” on page 697
Learn about the Message Queue Interface (MQI) components.

“Connecting to and disconnecting from a queue manager” on page 709

To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Opening and closing objects" on page 716
This information provides an insight into opening and closing IBM MQ objects.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Getting messages from a queue" on page 741
Use this information to learn about getting messages from a queue.

"Inquiring about and setting object attributes" on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

"Committing and backing out units of work" on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Working with the MQI and clusters" on page 852
There are special options on calls and return codes that relate to clustering.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

z/OS Writing IMS applications using IBM MQ

There are further considerations when using IBM MQ in IMS applications These include which MQ API calls can be used and the mechanism used for syncpoint.

Use the following links to find out more about writing IMS applications on IBM MQ for z/OS:

- "Syncpoints in IMS applications" on page 67
- "MQI calls in IMS applications" on page 68

Restrictions

There are restrictions on which IBM MQ API calls can used by an application using the IMS adapter.

The following IBM MQ API calls are not supported within an application using the IMS adapter:

- MQCB
- MQCB_FUNCTION
- MQCTL

Related concepts

"Writing IMS bridge applications" on page 71
This topic contains information about writing applications to use the IBM MQ - IMS bridge.

z/OS *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 ROLB call in the IMS documentation.

The queue manager is a participant in a two-phase commit protocol; the IMS syncpoint manager is the coordinator.

All open handles are closed by the IMS adapter at a syncpoint (except in a batch or non-message driven BMP environment). This is because a different user could initiate the next unit of work and IBM MQ

security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made.

However, in a Wait-for-Input (WFI) or pseudo Wait-for-Input (PWFI) environment IMS does not notify IBM MQ to close the handles until either the next message arrives or a QC status code is returned to the application. If the application is waiting in the IMS region and any of these handles belong to triggered queues, triggering will not occur because the queues are open. For this reason, applications running in a WFI or PWFI environment should explicitly MQCLOSE the queue handles before doing the GU to the IOPCB for the next message.

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.

z/OS *MQI calls in IMS applications*

Use this information to learn about the use of MQI calls on Server applications and Enquiry applications.

This section covers the use of MQI calls in the following types of IMS applications:

- "Server applications" on page 68
- "Inquiry applications" on page 70

Server applications

Here is an outline of the MQI server application model:

```
Initialize/Connect
.
Open queue for input shared
.
Get message from IBM 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 IBM 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 IBM MQ:

```
main()
----
Call InitIMS
If IMS initialization successful
Call InitMQM
If IBM 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 IBM 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 IBM 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 IBM MQ IMS trigger monitor program (CSQQTRMN) to schedule the MPP transaction when there are messages on the IBM 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.

Inquiry applications

A typical IBM MQ application initiating an inquiry or update works as follows:

- Gather data from the user
- Put one or more IBM MQ messages
- Get the reply messages (you might have to wait for them)
- Provide a response to the user

Because messages put on to IBM MQ queues do not become available to other IBM 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 Intersystem Communication (ISC)* 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 IBM MQ queue
.
Switch to second IBM 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 IBM MQ queue
.
Return results to originator
.
END
```

z/OS Writing IMS bridge applications

This topic contains information about writing applications to use the IBM MQ - IMS bridge.

For information about the IBM MQ - IMS bridge, see The IMS bridge.

Use the following links to find out more about writing IMS bridge applications on IBM MQ for z/OS:

- "How the IMS bridge deals with messages" on page 71
- "Writing IMS transaction programs through IBM MQ" on page 878

Related concepts

"Writing IMS applications using IBM MQ" on page 67
There are further considerations when using IBM MQ in IMS applications These include which MQ API calls can be used and the mechanism used for syncpoint.

z/OS *How the IMS bridge deals with messages*

When you use the IBM 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 IBM 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.

The IBM MQ-IMS bridge requires exclusive input access (MQOO_INPUT_EXCLUSIVE) to the bridge queue if it is defined with QSGDISP(QMGR), or if it is defined with QSGDISP(SHARED) together with the NOSHARE option.

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 IBM MQ connects to IMS, and is described in Application access control for the IMS bridge. This enables a pseudo signon to be implemented.

The IBM MQ - IMS bridge accepts the following types of message:

- Messages containing IMS transaction data and an MQIIH structure (described in MQIIH):

```
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>]
```

IBM 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 IBM 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 MQIIH), 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 IBM MQ - IMS bridge uses one or two Tpipes for each IBM 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 IBM 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.

See the following topics for more information about how the IBM MQ - IMS bridge deals with messages:

- “Mapping IBM MQ messages to IMS transaction types” on page 73
- “If the message cannot be put to the IMS queue” on page 73
- “IMS bridge feedback codes” on page 74
- “The MQMD fields in messages from the IMS bridge” on page 74
- “The MQIIH fields in messages from the IMS bridge” on page 75
- “Reply messages from IMS” on page 76
- “Using alternate response PCBs in IMS transactions” on page 76
- “Sending unsolicited messages from IMS” on page 76
- “Message segmentation” on page 77
- “Data conversion for messages to and from the IMS bridge” on page 77

Related concepts

“Writing IMS transaction programs through IBM MQ” on page 878

The coding required to handle IMS transactions through IBM 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.

z/OS *Mapping IBM MQ messages to IMS transaction types*

A table describing the mapping of IBM MQ messages to IMS transaction types.

Table 4. How IBM MQ messages map to IMS transaction types

IBM MQ message type	Commit-then-send (mode 0) - uses synchronized IMS Tpipes	Send-then-commit (mode 1) - uses non-synchronized IMS Tpipes
Persistent IBM MQ messages	• Recoverable full function transactions • Unrecoverable transactions are rejected by IMS	• Fastpath transactions • Conversational transactions • Full function transactions
Nonpersistent IBM MQ messages	• Unrecoverable full function transactions • Recoverable transactions are permitted with IMS V8 and APAR PQ61404 and all later versions of IMS	• Fastpath transactions • Conversational transactions • Full function transactions

Note: IMS commands cannot use persistent IBM MQ messages with commit mode 0. See Commit mode (commitMode) for more information.

z/OS *If the message cannot be put to the IMS queue*

Learn about actions to take 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 IBM 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, IBM MQ sends an error message to the system console, the message includes the IMS sense code, and the IBM MQ message is put to the dead-letter queue. If the IMS sense code is 001A, IMS sends an IBM MQ message containing the reason for the failure to the reply-to queue.

  Note: In the circumstances listed previously, if IBM MQ cannot put the message to the dead-letter queue for any reason, the message is returned to the originating IBM 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 IBM MQ subsystem
- If the message is rejected by IMS for anything other than a message error, the IBM MQ message is returned to the originating queue, IBM 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.

z/OS *IMS bridge feedback codes*

IMS sense codes are typically output in hexadecimal format in IBM MQ console messages such as CSQ2001I (for example, sense code 0x001F). IBM 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, or 600 through 855 for NACK sense code 0x001A. 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 IBM MQ *Feedback* code.
3. The IMS-OTMA sense code 0x001A, decimal 26 is a special case. A *Feedback* code in the range 600-855 is generated.
   a. The IMS-OTMA reason code is converted from a hexadecimal number to a decimal number.
   b. 600 is added to the number resulting from the calculation in a, giving the IBM MQ *Feedback* code.

For information about IMS-OTMA sense codes, see OTMA sense codes for NAK messages.

z/OS *The MQMD fields in messages from the IMS bridge*

Learn about 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

z/OS *The MQIIH fields in messages from the IMS bridge*

Learn about 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. In versions of IMS prior to V14, this field is always nulls if not in conversation. From IMS V14 onwards, this field may be set by IMS even if not in conversation.

TranState
"C" if in conversation, otherwise blank

CommitMode
Commit mode from OTMA header ("0" or "1")

SecurityScope
Blank

Reserved
Blank

z/OS *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 IBM 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 acknowledgment 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.

z/OS *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 Using OTMA exits in IMS and The pre-routing exit DFSYPRX0 for information about these exit programs.

Unless action is taken in the exits, all output from IMS transactions initiated from an IBM MQ queue manager, whether to the IOPCB or to an ALTPCB, will be returned to the same queue manager.

z/OS *Sending unsolicited messages from IMS*

To send messages from IMS to an IBM 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 pre-routing exit DFSYPRX0 and The destination resolution user exit for information about these exit programs.

Note: The IBM 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.

z/OS *Message segmentation*

You can define IMS transactions as expecting single- or multi-segment input.

The originating IBM 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 IBM MQ message sent with a single MQPUT.

The maximum length of an LLZZ-data segment is defined by IMS/OTMA (32767 bytes). The total IBM 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 IBM 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.

z/OS *Data conversion for messages to and from the IMS bridge*

The data conversion is performed by either the distributed queuing facility (which may call any necessary exits) or by the intra group queuing agent (which does not support the use of exits) when it puts a message to a destination queue that has XCF information defined for its storage class. The data conversion does not occur when a message is delivered to a queue by publish/subscribe.

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 IBM MQ - IMS bridge from any IBM MQ platform.

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 IBM 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 947 for detailed information about data conversion in general.

Sending messages to the IBM 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 can 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 IBM 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.

Developing JMS/Jakarta Messaging and Java applications

IBM MQ provides three Java language interfaces: IBM MQ classes for Jakarta Messaging, IBM MQ classes for JMS, and IBM MQ classes for Java.

About this task

JMS 3.0 **IBM MQ classes for Jakarta Messaging**

IBM MQ classes for Jakarta Messaging is a Jakarta Messaging provider that implements the Jakarta Messaging interfaces for IBM MQ as the messaging system. The Jakarta Connectors Architecture provides a standard way of connecting applications running in a Jakarta EE environment to an Enterprise Information System (EIS) such as IBM MQ or Db2.

For more information, see "Why should I use IBM MQ classes for Jakarta Messaging?" on page 80 and "Accessing IBM MQ from Java - Choice of API" on page 82.

JMS 2.0 **IBM MQ classes for JMS**

IBM MQ classes for JMS is a JMS provider that implements the JMS interfaces for IBM MQ as the messaging system. The Java Platform, Enterprise Edition Connector Architecture (JCA) provides a standard way of connecting applications running in a Java EE environment to an Enterprise Information System (EIS) such as IBM MQ or Db2.

For more information, see "Why should I use IBM MQ classes for JMS?" on page 81 and "Accessing IBM MQ from Java - Choice of API" on page 82.

IBM MQ classes for Java

IBM MQ classes for Java enable you to use IBM MQ in a Java environment. IBM MQ classes for Java allow a Java application to connect to IBM MQ as an IBM MQ client, or connect directly to an IBM MQ queue manager.

IBM MQ classes for Java encapsulates the Message Queue Interface (MQI), the native IBM MQ API, and uses the same object model as other object-oriented interfaces, whereas IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging implement Java messaging interfaces from Oracle and the Java Community Process respectively.

For more information, see "Why should I use IBM MQ classes for Java?" on page 336 and "Accessing IBM MQ from Java - Choice of API" on page 82.

Note:

Stabilized IBM will make no further enhancements to the IBM MQ classes for Java and they are functionally stabilized at the level shipped in IBM MQ 8.0. Existing applications that use the IBM MQ classes for Java continue to be fully supported, but new features will not be added and requests for enhancements will be rejected. Fully supported means that defects will be fixed together with any changes necessitated by changes to IBM MQ System Requirements.

The IBM MQ classes for Java are not supported in IMS.

The IBM MQ classes for Java are not supported in WebSphere® Liberty. They must not be used with either the IBM MQ Liberty messaging feature, or with the generic JCA support. For more information, see Using WebSphere MQ Java Interfaces in J2EE/JEE Environments.

Using IBM MQ classes for JMS/Jakarta Messaging

IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 continues to support JMS 2.0 for existing applications. It is not supported to use both the JMS 2.0 API and the Jakarta Messaging 3.0 API in the same application.

Note: For Jakarta Messaging 3.0, control of the JMS specification moves from Oracle to the Java Community Process. However, Oracle retains control of the "javax" name, which is used in other Java technologies that have not moved to the Java Community Process. So, while Jakarta Messaging 3.0 is functionally equivalent to JMS 2.0 there are some differences in naming:

- The official name for version 3.0 is Jakarta Messaging rather than Java Message Service.
- The package and constant names are prefixed with `jakarta` rather than `javax`. For example, in JMS 2.0 the initial connection to a messaging provider is a `javax.jms.Connection` object, and in Jakarta Messaging 3.0 it is a `jakarta.jms.Connection` object.

JMS 2.0 The javax.jms packages define the JMS interfaces, and a JMS provider implements these interfaces for a specific messaging product. IBM MQ classes for JMS is a JMS provider that implements the JMS interfaces for IBM MQ.

JMS 3.0 The jakarta.jms packages define the Jakarta Messaging interfaces, and a Jakarta Messaging provider implements these interfaces for a specific messaging product. IBM MQ classes for Jakarta Messaging is a Jakarta Messaging provider that implements the Jakarta Messaging interfaces for IBM MQ.

The JMS and Jakarta Messaging specifications expect ConnectionFactory and Destination objects to be administered objects. An administrator creates and maintains administered objects in a central repository, and a JMS or Jakarta Messaging application retrieves these objects using the Java Naming Directory Interface (JNDI).

JMS 2.0 For JMS 2.0, an administrator can use the IBM MQ JMS administration tool **JMSAdmin**, or IBM MQ Explorer, to create and maintain administered objects in a central repository.

JMS 3.0 For Jakarta Messaging 3.0, you cannot administer JNDI using IBM MQ Explorer. JNDI administration is supported by the Jakarta Messaging 3.0 variant of **JMSAdmin**, which is **JMS30Admin**.

Because JMS and Jakarta Messaging share much in common, further references to JMS in this topic can be taken as referring to both. Any differences are highlighted as necessary.

IBM MQ classes for JMS also provides two sets of extensions to the JMS API. The main focus of these extensions concerns creating and configuring connection factories and destinations dynamically at run time, but the extensions also provide function that is not directly related to messaging, such as function for problem determination.

The IBM MQ JMS extensions

IBM MQ classes for JMS contains extensions that are implemented in objects such as MQConnectionFactory, MQQueue, and MQTopic objects. These objects have properties and methods that are specific to IBM MQ. The objects can be administered objects, or an application can create the objects dynamically at run time. These extensions are termed the IBM MQ JMS extensions.

The IBM JMS extensions

IBM MQ classes for JMS also provides a more generic set of extensions to the JMS API, which are not specific to IBM MQ as the messaging system or Java as the programming language used. These extensions are termed the IBM JMS extensions and have the following broad objectives:

- To provide a greater level of consistency across IBM JMS providers.
- To make it easier to write a bridge application between two IBM messaging systems.
- To make it easier to port an application from one IBM JMS provider to another.

The extensions provide function that is similar to that provided in IBM MQ Message Service Client (XMS) for C/C++ and IBM MQ Message Service Client (XMS) for .NET.

Related concepts

IBM MQ Java language interfaces

Related tasks

"Writing IBM MQ classes for JMS/Jakarta Messaging applications" on page 134

After a brief introduction to the JMS model, this section provides detailed guidance on how to write IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging applications.

JMS 3.0 Why should I use IBM MQ classes for Jakarta Messaging?

Using IBM MQ classes for Jakarta Messaging has a number of advantages including being able to reuse any existing Jakarta Messaging skills in your organization, and applications being more independent from the Jakarta Messaging provider and the underlying IBM MQ configuration.

Summary of advantages of using IBM MQ classes for Jakarta Messaging

Using IBM MQ classes for Jakarta Messaging allows you to reuse existing Jakarta Messaging skills and provide application independence.

- You can reuse Jakarta Messaging skills.

  IBM MQ classes for Jakarta Messaging is a Jakarta Messaging provider that implements the Jakarta Messaging interfaces for IBM MQ as the messaging system. If your organization is new to IBM MQ, but already has Jakarta Messaging (or JMS) application development skills, you might find it easier to use the familiar Jakarta Messaging API to access IBM MQ resources rather than one of the other APIs provided with IBM MQ.

- Jakarta Messaging is an integral part of Jakarta EE.

  Jakarta Messaging is the natural API to use for messaging on the Jakarta EE platform. Every application server that is Jakarta EE compliant must include a Jakarta Messaging provider. You can use Jakarta Messaging in application clients, servlets, Java Server Pages (JSPs), enterprise Java beans (EJBs), and message driven beans (MDBs). Note in particular that Jakarta EE applications use MDBs to process messages asynchronously, and all messages are delivered to MDBs as Jakarta Messaging messages.

- Connection factories and destinations can be stored as Jakarta Messaging administered objects in a central repository rather than being hard-coded into an application.

  An administrator can create and maintain Jakarta Messaging administered objects in a central repository, and IBM MQ classes for Jakarta Messaging applications can retrieve these objects by using the Java Naming Directory Interface (JNDI). Jakarta Messaging connection factories and destinations encapsulate IBM MQ-specific information such as queue manager names, channel names, connection options, queue names, and topic names. If connection factories and destinations are stored as administered objects, this information is not hard-coded into an application. This arrangement therefore provides the application with a degree of independence from the underlying IBM MQ configuration.

- Jakarta Messaging is an industry standard API that can provide application portability.

  A Jakarta Messaging application can use JNDI to retrieve connection factories and destinations that are stored as administered objects, and use only the interfaces that are defined in the `jakarta.jms` (Jakarta Messaging 3.0) package to perform messaging operations. The application is then entirely independent of any Jakarta Messaging provider, such as IBM MQ classes for Jakarta Messaging, and can be ported from one Jakarta Messaging provider to another without any change to the application.

  If JNDI is not available in a particular application environment, a IBM MQ classes for Jakarta Messaging application can use extensions to the Jakarta Messaging API to create and configure connection factories and destinations dynamically at run time. The application is then completely self-contained, but is tied to IBM MQ classes for Jakarta Messaging as the Jakarta Messaging provider.

- Bridge applications might be easier to write by using Jakarta Messaging.

A bridge application is an application that receives messages from one messaging system and sends them to another messaging system. Writing a bridge application can be complicated by using product-specific APIs and message formats. Instead, you can write a bridge application by using two Jakarta Messaging providers, one for each messaging system. The application then uses only one API, the Jakarta Messaging API, and processes only Jakarta Messaging messages.

Deployable environments

To provide integration with a Jakarta EE application server, the Jakarta EE standards require messaging providers to supply a resource adapter. Following the Jakarta Connectors Architecture specification, IBM MQ provides a resource adapter that uses Jakarta Messaging to provide messaging functions within any certified Jakarta EE environment. For more information, see "Liberty and the IBM MQ resource adapter" on page 425.

Note: WebSphere Application Server traditional does not currently support Jakarta EE.

Outside of the Jakarta EE environment, OSGi and JAR files are provided, making it easier for you to obtain just the IBM MQ classes for Jakarta Messaging. These JAR files are more readily deployable either stand-alone or within software management frameworks such as Maven. For more information see "Obtaining the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging separately" on page 122.

Related concepts

IBM MQ classes for Jakarta Messaging: an overview

"Accessing IBM MQ from Java - Choice of API" on page 82

IBM MQ provides three Java language interfaces.

JMS 2.0 Why should I use IBM MQ classes for JMS?

Using IBM MQ classes for JMS has a number of advantages including being able to reuse any existing JMS skills in your organization, and applications being more independent from the JMS provider and the underlying IBM MQ configuration.

Summary of advantages of using IBM MQ classes for JMS

Using IBM MQ classes for JMS allows you to reuse existing JMS skills and provide application independence.

Note: JMS 2.0 has been superseded by Jakarta Messaging. IBM MQ classes for JMS continues to support the JMS 2.0 standard, but future enhancements to Java messaging will only emerge in Jakarta Messaging, hence in the IBM MQ classes for Jakarta Messaging. IBM MQ classes for JMS are only recommended for maintaining and extending existing JMS 2.0 applications. IBM MQ classes for Jakarta Messaging should be the preferred technology for new development.

- You can reuse JMS skills.

  IBM MQ classes for JMS is a JMS provider that implements the JMS interfaces for IBM MQ as the messaging system. If your organization is new to IBM MQ, but already has JMS application development skills, you might find it easier to use the familiar JMS API to access IBM MQ resources rather than one of the other APIs provided with IBM MQ.

- JMS is an integral part of Java Platform, Enterprise Edition (Java EE).

  JMS is the natural API to use for messaging on the Java EE platform. Every application server that is Java EE compliant must include a JMS provider. You can use JMS in application clients, servlets, Java Server Pages (JSPs), enterprise Java beans (EJBs), and message driven beans (MDBs). Note in particular that Java EE applications use MDBs to process messages asynchronously, and all messages are delivered to MDBs as JMS messages.

- Connection factories and destinations can be stored as JMS administered objects in a central repository rather than being hard-coded into an application.

An administrator can create and maintain JMS administered objects in a central repository, and IBM MQ classes for JMS applications can retrieve these objects by using the Java Naming Directory Interface (JNDI). JMS connection factories and destinations encapsulate IBM MQ-specific information such as queue manager names, channel names, connection options, queue names, and topic names. If connection factories and destinations are stored as administered objects, this information is not hard-coded into an application. This arrangement therefore provides the application with a degree of independence from the underlying IBM MQ configuration.

- JMS is an industry standard API that can provide application portability.

  A JMS application can use JNDI to retrieve connection factories and destinations that are stored as administered objects, and use only the interfaces that are defined in the `javax.jms` package to perform messaging operations. The application is then entirely independent of any JMS provider, such as IBM MQ classes for JMS, and can be ported from one JMS provider to another without any change to the application.

  If JNDI is not available in a particular application environment, an IBM MQ classes for JMS application can use extensions to the JMS API to create and configure connection factories and destinations dynamically at run time. The application is then completely self-contained, but is tied to IBM MQ classes for JMS as the JMS provider.

- Bridge applications might be easier to write by using JMS.

  A bridge application is an application that receives messages from one messaging system and sends them to another messaging system. Writing a bridge application can be complicated by using product-specific APIs and message formats. Instead, you can write a bridge application by using two JMS providers, one for each messaging system. The application then uses only one API, the JMS API, and processes only JMS messages.

Deployable environments

To provide integration with a Java EE application server, the Java EE standards require messaging providers to supply a resource adapter. Following the Java EE Connector Architecture (JCA) specification, IBM MQ provides a resource adapter that uses JMS to provide messaging functions within any certified Java EE environment.

While it has been possible to use the IBM MQ classes for Java inside Java EE, this API is not engineered or optimized for this purpose. For more information about IBM MQ classes for Java considerations within Java EE, see “Running IBM MQ classes for Java applications within Java EE” on page 337.

Outside of the Java EE environment, OSGi and JAR files are provided, making it easier for you to obtain just the IBM MQ classes for JMS. These JAR files are more readily deployable either stand-alone or within software management frameworks such as Maven. For more information see “Obtaining the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging separately” on page 122.

Related concepts

IBM MQ classes for Jakarta Messaging: an overview

“Why should I use IBM MQ classes for Jakarta Messaging?” on page 80
Using IBM MQ classes for Jakarta Messaging has a number of advantages including being able to reuse any existing Jakarta Messaging skills in your organization, and applications being more independent from the Jakarta Messaging provider and the underlying IBM MQ configuration.

“Accessing IBM MQ from Java - Choice of API” on page 82
IBM MQ provides three Java language interfaces.

Accessing IBM MQ from Java - Choice of API

IBM MQ provides three Java language interfaces.

- IBM MQ classes for Jakarta Messaging
- IBM MQ classes for JMS
- IBM MQ classes for Java

IBM MQ classes for Jakarta Messaging

IBM MQ classes for Jakarta Messaging allows applications written using the Jakarta Messaging 3.0 APIs to utilise IBM MQ as a messaging provider.

Jakarta Messaging is the strategic direction for messaging in Java applications.

Jakarta Messaging 3.0 is functionally equivalent to JMS 2.0, so for more information see "Using IBM MQ classes for JMS/Jakarta Messaging" on page 79.

IBM MQ classes for JMS

IBM MQ classes for JMS allows applications written using the JMS 2.0 APIs to utilise IBM MQ as a messaging provider.

As Jakarta Messaging supersedes JMS, IBM MQ classes for JMS is recommended for use in existing applications or in environments (for example, WebSphere Application Server) that do not support Jakarta Messaging.

It is not supported to use both IBM MQ classes for Jakarta Messaging and IBM MQ classes for JMS in the same application.

For more information see "Using IBM MQ classes for JMS/Jakarta Messaging" on page 79.

IBM MQ classes for Java

Stabilized The other API that Java applications can use to access IBM MQ resources is IBM MQ classes for Java, which provides an IBM MQ-oriented API for programs to use IBM MQ as a messaging provider. However, IBM MQ classes for Java is functionally stabilized at the level shipped in IBM MQ 8.0. For more information, see "Why should I use IBM MQ classes for Java?" on page 336. Although existing applications that use IBM MQ classes for Java continue to be fully supported, new applications should use IBM MQ classes for Jakarta Messaging.

Common features of IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging

IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging provide access to both the point-to-point and publish/subscribe messaging features of IBM MQ. As well as sending JMS messages that provide support for the JMS standard messaging model, applications can also send and receive messages without additional headers and so can inter-operate with other IBM MQ applications, for example, C MQI applications. Full control of the MQMD and MQ message payloads is available.

Further IBM MQ features such as message streaming, asynchronous put and report messages are also available.

Using the supplied PCF helper classes, IBM MQ PCF administration messages can be sent and received through the JMS API and can be used to administer queue managers.

Features that have recently been added to IBM MQ, such as asynchronous consume and automatic reconnection, are not available in the IBM MQ classes for Java, but are available in the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

Requesting enhancements

If you need access to IBM MQ features that are not available through IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging, you can raise an idea.

IBM can then advise whether the implementation is possible in the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging implementation, or whether there is a best practice that can be followed.

For additional messaging features, as IBM is a contributor to the open standard, these features can be raised as part of the JCP process. These would only apply to Jakarta Messaging.

Related information

Welcome to the IBM Ideas Portal

JMS Java Specification Review Process

Using JMS to send PCF messages

JMS 3.0 Prerequisites for IBM MQ classes for Jakarta Messaging

This topic tells you what you need to know before using IBM MQ classes for Jakarta Messaging. To develop and run IBM MQ classes for Jakarta Messaging applications, you need certain software components as prerequisites.

For information about the prerequisites for IBM MQ classes for Jakarta Messaging, see System Requirements for IBM MQ.

To develop IBM MQ classes for Jakarta Messaging applications, you need a Java SE Software Development Kit (SDK). for details of the JDKs supported by your operating system, see System Requirements for IBM MQ.

To run IBM MQ classes for Jakarta Messaging applications, you need the following software components:

- An IBM MQ queue manager.
- A Java runtime environment (JRE), for each system on which you run applications.
- IBM i For IBM i, Qshell, which is option 30 of the operating system.
- z/OS For z/OS, z/OS UNIX System Services (z/OS UNIX).

The IBM JSSE provider includes a FIPS certified cryptographic provider, so can be programmatically configured for FIPS 140-2 compliance ready for immediate use. Therefore, FIPS 140-2 compliance can be supported directly from IBM MQ classes for Jakarta Messaging.

Oracle's JSSE provider can have a FIPS certified cryptographic provider that is configured into it, but this is not ready for immediate use and is not available for programmatic configuration. Therefore, in this case, IBM MQ classes for Jakarta Messaging cannot enable FIPS 140-2 compliance directly. You might be able to manually enable such compliance, but IBM cannot currently provide guidance on this.

You can use Internet Protocol version 6 (IPv6) addresses in your IBM MQ classes for Jakarta Messaging applications if IPv6 addresses are supported by your Java virtual machine (JVM) and the TCP/IP implementation on your operating system. The IBM MQ Jakarta Messaging administration tool, **JMS30Admin**, also accepts IPv6 addresses. For more information about this tool, see Configuring JMS and Jakarta Messaging objects using the administration tools.

The IBM MQ JMS administration tool and IBM MQ Explorer use the Java Naming Directory Interface (JNDI) to access a directory service, which stores administered objects. IBM MQ classes for Jakarta Messaging applications can also use JNDI to retrieve administered objects from a directory service.

Note: For Jakarta Messaging 3.0, you cannot administer JNDI using IBM MQ Explorer. JNDI administration is supported by the Jakarta Messaging 3.0 variant of **JMSAdmin**, which is **JMS30Admin**.

A service provider is code that provides access to a directory service by mapping JNDI calls to the directory service. A file system service provider in the files `fscontext.jar` and `providerutil.jar` is supplied with IBM MQ classes for Jakarta Messaging. The file system service provider provides access to a directory service based on the local file system.

If you intend to use a directory service based on an LDAP server, you must install and configure an LDAP server, or have access to an existing LDAP server. In particular, you must configure the LDAP server to store Java objects. For information about how to install and configure your LDAP server, see the documentation that is supplied with the server.

JMS 2.0 Prerequisites for IBM MQ classes for JMS

This topic tells you what you need to know before using IBM MQ classes for JMS. To develop and run IBM MQ classes for JMS applications, you need certain software components as prerequisites.

For information about the prerequisites for IBM MQ classes for JMS, see System Requirements for IBM MQ.

To develop IBM MQ classes for JMS applications, you need a Java SE Software Development Kit (SDK). for details of the JDKs supported by your operating system, see System Requirements for IBM MQ.

To run IBM MQ classes for JMS applications, you need the following software components:

- An IBM MQ queue manager.
- A Java runtime environment (JRE), for each system on which you run applications.
- IBM i For IBM i, Qshell, which is option 30 of the operating system.
- z/OS For z/OS, z/OS UNIX System Services (z/OS UNIX).

The IBM JSSE provider includes a FIPS certified cryptographic provider, so can be programmatically configured for FIPS 140-2 compliance ready for immediate use. Therefore, FIPS 140-2 compliance can be supported directly from IBM MQ classes for Java and IBM MQ classes for JMS.

Oracle's JSSE provider can have a FIPS certified cryptographic provider that is configured into it, but this is not ready for immediate use and is not available for programmatic configuration. Therefore, in this case, IBM MQ classes for Java and IBM MQ classes for JMS cannot enable FIPS 140-2 compliance directly. You might be able to manually enable such compliance, but IBM cannot currently provide guidance on this.

You can use Internet Protocol version 6 (IPv6) addresses in your IBM MQ classes for JMS applications if IPv6 addresses are supported by your Java virtual machine (JVM) and the TCP/IP implementation on your operating system. The IBM MQ JMS administration tool (see Configuring JMS objects using the administration tool) also accepts IPv6 addresses.

The IBM MQ JMS administration tool and IBM MQ Explorer use the Java Naming Directory Interface (JNDI) to access a directory service, which stores administered objects. IBM MQ classes for JMS applications can also use JNDI to retrieve administered objects from a directory service. A service provider is code that provides access to a directory service by mapping JNDI calls to the directory service. A file system service provider in the files `fscontext.jar` and `providerutil.jar` is supplied with IBM MQ classes for JMS. The file system service provider provides access to a directory service based on the local file system.

If you intend to use a directory service based on an LDAP server, you must install and configure an LDAP server, or have access to an existing LDAP server. In particular, you must configure the LDAP server to store Java objects. For information about how to install and configure your LDAP server, see the documentation that is supplied with the server.

Installing and configuring IBM MQ classes for JMS/Jakarta Messaging

This section describes the directories and files that are created when you install IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging, and tells you how to configure IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging after installation.

Related concepts

“Using the IBM MQ resource adapter” on page 420

The resource adapter allows applications that are running in an application server to access IBM MQ resources. It supports inbound and outbound communication.

What is installed for IBM MQ classes for JMS

A number of files and directories are created when you install IBM MQ classes for JMS. On Windows, some configuration is performed during installation by automatically setting environment variables. On

other platforms, and in certain Windows environments, you must set environment variables before you can run IBM MQ classes for JMS applications.

For most operating systems, the IBM MQ classes for JMS are installed as an optional component when you install IBM MQ.

For more information about installing IBM MQ, see:

Multi Installing IBM MQ

z/OS Installing IBM MQ for z/OS

Important: Apart from the relocatable JAR files described in "IBM MQ classes for JMS/Jakarta Messaging relocatable JAR files" on page 87, copying the IBM MQ classes for JMS JAR files or native libraries to other machines, or to a different location on a machine where the IBM MQ classes for JMS have been installed, is not supported.

Installation directories

Table 5 on page 86 shows where the IBM MQ classes for JMS files are installed on each platform.

Table 5. IBM MQ classes for JMS installation directories

Platform	Directory
Linux AIX AIX and Linux	*MQ_INSTALLATION_PATH*`/java`
Windows Windows	*MQ_INSTALLATION_PATH*`\java`
IBM i IBM i	`/QIBM/ProdData/mqm/java`
z/OS z/OS	*MQ_INSTALLATION_PATH*`/java`

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

The installation directory includes the following content:

- The IBM MQ classes for JMS JAR files, including the relocatable JAR files, which are in the *MQ_INSTALLATION_PATH*`\java\lib` directory.
- The IBM MQ native libraries, which are used by applications that use the Java Native Interface.

  The 32-bit native libraries are installed into the *MQ_INSTALLATION_PATH*`\java\lib` directory and the 64-bit native libraries can be found in the *MQ_INSTALLATION_PATH*\java\lib64 directory.

  For more information about the IBM MQ native libraries, see "Configuring the Java Native Interface (JNI) libraries" on page 92.
- Additional scripts that are described in "Scripts provided with IBM MQ classes for JMS/Jakarta Messaging" on page 118. These scripts are in the *MQ_INSTALLATION_PATH*`\java\bin` directory.
- The specifications of the IBM MQ classes for JMS API. The Javadoc tool has been used to generate the HTML pages that contain the specifications of the API.

  The HTML pages are in the *MQ_INSTALLATION_PATH*`\java\doc\WMQJMSClasses` directory:

  – ALW On AIX, Linux, and Windows, this subdirectory contains the individual HTML pages.

  – IBM i On IBM i, the HTML pages are in a file called `wmqjms_javadoc.jar`.

  – z/OS On z/OS, the HTML pages are in a file called `wmqjms_javadoc.jar`.
- Support for OGSi. OSGi bundles are installed in the `java\lib\OSGi` directory and described in "Support for OSGi with IBM MQ classes for JMS" on page 119.

- The IBM MQ resource adapter, which can be deployed into any Java Platform, Enterprise Edition 7 (Java EE 7) or Jakarta EE compliant application server.

  The IBM MQ resource adapter is in the *MQ_INSTALLATION_PATH*`\java\lib\jca` directory; for more information, see "Using the IBM MQ resource adapter" on page 420

- Windows On Windows, symbols that can be used for debugging are installed in the *MQ_INSTALLATION_PATH*`\java\lib\symbols` directory.

The installation directory also includes some files that belong to other IBM MQ components.

Sample applications

JMS 2.0 Some sample applications are supplied with IBM MQ classes for JMS. Table 6 on page 87 shows where the sample applications are installed on each platform.

JMS 3.0 For IBM MQ classes for Jakarta Messaging, new samples are being prepared.

JMS 2.0

Table 6. Samples directories for IBM MQ classes for JMS

Platform	Directory
Linux AIX AIX and Linux	*MQ_INSTALLATION_PATH*`/samp/jms`
Windows Windows	*MQ_INSTALLATION_PATH*`\tools\jms`
IBM i IBM i	`/QIBM/ProdData/mqm/java/samples/jms`
z/OS z/OS	*MQ_INSTALLATION_PATH*`/java/samples/jms`

In this table, *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

After installation, you might need to perform some configuration tasks to compile and run applications.

"Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90 describes the classpath that is required to run sample IBM MQ classes for JMS applications. This topic also describes additional JAR files that need to be referenced in special circumstances and the environment variables that you must set to run the scripts that are provided with IBM MQ classes for JMS.

To control properties, such as tracing and logging of an application, you need to provide a configuration properties file. The IBM MQ classes for JMS configuration properties file is described in "The IBM MQ classes for JMS/Jakarta Messaging configuration file" on page 94.

Related concepts

Problems in deploying the resource adapter

Related tasks

"Using the IBM MQ classes for JMS sample applications" on page 115
The IBM MQ classes for JMS sample applications provide an overview of the common features of the JMS API. You can use them to verify your installation and messaging server set up and to help you build your own applications.

IBM MQ classes for JMS/Jakarta Messaging relocatable JAR files
The relocatable JAR files can be moved to systems that need to run IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging.

Important:

- Apart from the relocatable JAR files described in Relocatable JAR files, copying the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging JAR files or native libraries to other machines, or to

a different location on a machine where the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging have been installed, is not supported.

- Do not include the relocatable JAR files within applications deployed into Java EE application servers, such as WebSphere Application Server or WebSphere Liberty. In these environments, the IBM MQ resource adapter should be deployed and used instead. Note that WebSphere Application Server embeds the IBM MQ resource adapter, so there is no need to deploy it manually into this environment.
- To avoid classloader conflicts, it is not recommended to bundle the relocatable JAR files within multiple applications inside the same Java runtime. In this scenario, make the IBM MQ relocatable JAR files available on the Java runtime's classpath.
- If you are bundling the relocatable JAR files within your applications, ensure that you include all prerequisite JAR files as described in Relocatable JAR files. You should also ensure that you have appropriate procedures to update the bundled JAR files as part of application maintenance, to ensure that the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging are still current and known issues are re-mediated.

Relocatable JAR files

Within an enterprise, the following files can be moved to systems that need to run IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging:

- `bcpkix-jdk15to18.jar` "4" on page 88
- V 9.4.0 `bcpkix-jdk18on.jar` "3" on page 88
- `bcprov-jdk15to18.jar` "4" on page 88
- V 9.4.0 `bcprov-jdk18on.jar` "3" on page 88
- `bcutil-jdk15to18.jar` "4" on page 88
- V 9.4.0 `bcutil-jdk18on.jar` "3" on page 88
- JMS 2.0 `com.ibm.mq.allclient.jar` "1" on page 88
- JMS 3.0 `com.ibm.mq.jakarta.client.jar` "2" on page 88
- `fscontext.jar`
- `jakarta.jms-api.jar`
- `jms.jar`
- `org.json.jar`
- `providerutil.jar`

Notes:

1. JMS 2.0 and JMS 1.1
2. Jakarta Messaging 3.0
3. From IBM MQ 9.4.0
4. Before IBM MQ 9.4.0

JMS JAR files

`jms.jar` contains the JMS 1.1 and JMS 2.0 interfaces - these are named `javax.jms.*`.

JMS 3.0 `jakarta.jms-api.jar` contains the Jakarta Messaging 3.0 interfaces - these are named `jakarta.jms.*`.

fscontext.jar and providerutil.jar

The `fscontext.jar` and `providerutil.jar` files are required if your application performs JNDI lookups using a file system context.

Bouncy Castle security provider and CMS support JAR files

The Bouncy Castle security provider and CMS support JAR files are required. For more information, see Support for non-IBM JREs with AMS.

V 9.4.0 The following JAR files are required:

- `bcpkix-jdk18on.jar`
- `bcprov-jdk18on.jar`
- `bcutil-jdk18on.jar`

org.json.jar

The `org.json.jar` file is required if your IBM MQ classes for JMS application uses a CCDT in JSON format.

com.ibm.mq.allclient.jar and com.ibm.mq.jakarta.client.jar

The files `com.ibm.mq.allclient.jar` and `com.ibm.mq.jakarta.client.jar` contain the IBM MQ classes for JMS, the IBM MQ classes for Jakarta Messaging, the IBM MQ classes for Java, and the PCF and Headers Classes. If you move these JAR file to a new location, make sure that you take steps to keep this new location maintained with new IBM MQ Fix Packs. Also, make sure that the use of the files is made known to IBM Support if you are getting an interim fix.

To determine the version of the files `com.ibm.mq.allclient.jar` and `com.ibm.mq.jakarta.client.jar`, use the following command:

JMS 3.0

```
java -jar com.ibm.mq.jakarta.client.jar
```

JMS 2.0

```
java -jar com.ibm.mq.allclient.jar
```

The following example shows some sample output from this command:

```
C:\Program Files\IBM\MQ_1\java\lib>java -jar com.ibm.mq.allclient.jar
Name:       Java Message Service Client
Version:    9.3.0.0
Level:      p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar

Name:       WebSphere MQ classes for Java Message Service
Version:    9.3.0.0
Level:      p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar

Name:       WebSphere MQ JMS Provider
Version:    9.3.0.0
Level:      p000-L140428.1 mqjbnd=p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar

Name:       Common Services for Java Platform, Standard Edition
Version:    9.3.0.0
Level:      p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar
```

Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging
Before you can compile and run IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging applications, the setting for your **CLASSPATH** environment variable must include the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging Java archive (JAR) file. Depending on your requirements, you might need to add other JAR files to your class path. To run the scripts provided with IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging, other environment variables must be set.

Before you begin

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. It is not supported to use both the Jakarta Messaging 3.0 API and the JMS 2.0 API in the same application. For more information, see Using IBM MQ classes for JMS/Jakarta Messaging.

Important: Setting the Java option *-Xbootclasspath* to include the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging is not supported.

About this task

To compile and run IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging applications, use the **CLASSPATH** setting for your platform and Java messaging version, as shown in the following tables. Alternatively, you can specify the class path on the **java** command instead of using the environment variable.

JMS 2.0 For IBM MQ classes for JMS, the setting includes the samples directory, so that you can compile and run the IBM MQ classes for JMS sample applications.

JMS 3.0 For IBM MQ classes for Jakarta Messaging, new samples are being prepared.

JMS 3.0

*Table 7. **CLASSPATH** settings for Jakarta Messaging 3.0 to compile and run IBM MQ classes for Jakarta Messaging applications*

Platform	CLASSPATH setting
AIX AIX	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.jakarta.client.jar:
Linux Linux	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.jakarta.client.jar:
IBM i IBM i	CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jakarta.client.jar:
Windows Windows	CLASSPATH= *MQ_INSTALLATION_PATH*\java\lib\com.ibm.mq.jakarta.client.jar;
z/OS z/OS	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.jakarta.client.jar;

JMS 2.0

*Table 8. **CLASSPATH** settings for JMS 2.0 to compile and run IBM MQ classes for JMS applications, including the sample applications*

Platform	CLASSPATH setting
AIX AIX	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.allclient.jar: *MQ_INSTALLATION_PATH*/samp/jms/samples:

*Table 8. **CLASSPATH** settings for JMS 2.0 to compile and run IBM MQ classes for JMS applications, including the sample applications (continued)*	
Platform	**CLASSPATH setting**
Linux Linux	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.allclient.jar: *MQ_INSTALLATION_PATH*/samp/jms/samples:
IBM i IBM i	CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mq.allclient.jar: /QIBM/ProdData/mqm/java/samples/jms/samples:
Windows Windows	CLASSPATH= *MQ_INSTALLATION_PATH*\java\lib\com.ibm.mq.allclient.jar; *MQ_INSTALLATION_PATH*\tools\jms\samples;
z/OS z/OS	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.allclient.jar: *MQ_INSTALLATION_PATH*/java/samples/jms/samples:

In these tables, *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

The manifest of the JAR file `com.ibm.mq.jakarta.client.jar` or `com.ibm.mq.allclient.jar` contains references to most of the other JAR files required by IBM MQ classes for JMS applications, and so you do not need to add these JAR files to your class path. These JAR files include those required by applications that use the Java Naming Directory Interface (JNDI) to retrieve administered objects from a directory service and by applications that use the Java Transaction API (JTA).

However, you must include additional JAR files in your class path in the following circumstances:

- If you are using channel exit classes that implement the channel exit interfaces defined in the `com.ibm.mq` package, instead of those defined in the `com.ibm.mq.exits` package, you must add the IBM MQ classes for Java JAR file, `com.ibm.mq.jar`, to your class path.
- If your application uses JNDI to retrieve administered objects from a directory service, you must also add the following JAR files to your class path:
  - `fscontext.jar`
  - `providerutil.jar`
- If your application uses the JTA, you must also add `jta.jar` to your class path.

Note: These additional JAR files are required only for compiling your applications, not for running them.

The scripts provided with IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging use the following environment variables:

MQ_JAVA_DATA_PATH
This environment variable specifies the directory for log and trace output.

MQ_JAVA_INSTALL_PATH
This environment variable specifies the directory where IBM MQ classes for JMS is installed.

MQ_JAVA_LIB_PATH
This environment variable specifies the directory where the IBM MQ classes for JMS libraries are stored, as shown in the previous tables.

Procedure

- Windows
  On Windows, after installing IBM MQ, run the command **setmqenv**.
  If you do not run this command first, the following error message might appear when you are issuing a **dspmqver** command:

```
AMQ8351: IBM MQ Java environment has not been configured
correctly, or the IBM MQ JRE feature has not been installed.
```

Note: This message is to be expected if you did not install the IBM MQ Java runtime environment (JRE) (see Additional Windows features prerequisite checking.

- Linux AIX

  On AIX and Linux systems, set the environment variables yourself:

  JMS 2.0 For JMS 2.0, use one of the following scripts to set the environment variables:

  – If you are using a 32-bit JVM, use the script `setjmsenv`.
  – If you are using a 64-bit JVM on an AIX or Linux system, use the script `setjmsenv64`.

  JMS 3.0 For Jakarta Messaging 3.0, use one of the following scripts to set the environment variables:

  – If you are using a 32-bit JVM, use the script `setjms30env`.
  – If you are using a 64-bit JVM, use the script `setjms30env64`.

  These scripts are in the *MQ_INSTALLATION_PATH*`/java/bin` directory, where *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

  You can use these scripts in a variety of ways. You can use the script as a basis for setting the required environment variables, as shown in the table, or add them to `.profile` using a text editor. If you have a non-typical setup, edit the script contents as necessary. Alternatively, you can run the script in every session from which JMS startup scripts are to be run. If you choose this option, you need to run the script in every shell window you start, during the JMS verification process:

  – JMS 2.0 For JMS 2.0, type `. ./setjmsenv` or `../setjmsenv64`.
  – JMS 3.0 For Jakarta Messaging 3.0, type `. ./setjms30env` or `. ./setjms30env64`.

  IBM i On IBM i, you must set the environment variable **QIBM_MULTI_THREADED** to Y. You can then run multithreaded applications in the same way that you run single threaded applications. For more information, see Setting up IBM MQ with Java and JMS.

Related tasks

“Using the IBM MQ classes for JMS sample applications” on page 115
The IBM MQ classes for JMS sample applications provide an overview of the common features of the JMS API. You can use them to verify your installation and messaging server set up and to help you build your own applications.

Related reference

“Scripts provided with IBM MQ classes for JMS/Jakarta Messaging” on page 118
A number of scripts are provided to assist with common tasks that need to be performed when using IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

Configuring the Java Native Interface (JNI) libraries
IBM MQ classes for JMS applications, that either connect to a queue manager using the bindings transport, or that connect to a queue manager using the client transport and use channel exit programs written in languages other than Java, need to be run in an environment that allows access to the Java Native Interface (JNI) libraries.

Before you begin

See Configuring the IBM MQ messaging provider with native libraries information for more information on using the WebSphere Application Server environment.

About this task

To set up this environment, you must configure the environment's library path so that the Java Virtual Machine (JVM) can load the mqjbnd library before you start the IBM MQ classes for JMS application.

IBM MQ provides two Java Native Interface (JNI) libraries:

mqjbnd
This library is used by applications that connect to a queue manager using the bindings transport. It provides the interface between the IBM MQ classes for JMS and the queue manager. The mqjbnd library installed with IBM MQ 9.4 can be used to connect to any IBM MQ 9.4 (or earlier) queue manager.

mqjexitstub02
The mqjexitstub02 library is loaded by the IBM MQ classes for JMS when an application connects to a queue manager using the client transport and uses a channel exit program written in a language other than Java.

On certain platforms, IBM MQ installs 32-bit and 64-bit versions of these JNI libraries. The location of the libraries for each platform is shown in Table 1.

Table 9. The location of the IBM MQ classes for JMS libraries for each platform

Platform	**Directory containing the IBM MQ classes for JMS libraries**
AIX AIX Linux Linux (POWER, x86-64 and zSeries s390x platforms)	*MQ_INSTALLATION_PATH*/java/lib (32-bit libraries) *MQ_INSTALLATION_PATH*/java/lib64 (64-bit libraries)
Windows Windows	*MQ_INSTALLATION_PATH*\java\lib (32-bit libraries) *MQ_INSTALLATION_PATH*\java\lib64 (64-bit libraries)
z/OS z/OS	*MQ_INSTALLATION_PATH*/java/lib (31-bit and 64-bit libraries)

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Note: z/OS On z/OS, you can use either a 31-bit or 64-bit Java Virtual Machine (JVM). You do not have to specify which JNI libraries to use; IBM MQ classes for JMS can determine for itself which JNI libraries to load.

Procedure

1. Configure the JVM's **`java.library.path`** property, which can be done in two ways:
   - By specifying the JVM argument as shown in the following example:

     ```
     -Djava.library.path=path_to_library_directory
     ```

     Linux For example, for a 64-bit JVM on Linux for a default location installation, specify:

     ```
     -Djava.library.path=/opt/mqm/java/lib64
     ```

   - By configuring the shell's environment such that the JVM will set up its own `java.library.path`. This path varies by platform and by the location in which you installed IBM MQ. For example, for a 64-bit JVM and a default IBM MQ installation location, you can use the following settings:


```
export LIBPATH=/usr/mqm/java/lib64:$LIBPATH
```


```
export LD_LIBRARY_PATH=/opt/mqm/java/lib64:$LD_LIBRARY_PATH
```


```
set PATH=C:\Program Files\IBM\MQ\java\lib64;%PATH%
```

An example of the exception stack that you see when the environment has not been configured correctly is as follows:

```
Caused by: com.ibm.mq.jmqi.local.LocalMQ$4: CC=2;RC=2495;
AMQ8598: Failed to load the WebSphere MQ native JNI library: 'mqjbnd'.
    at com.ibm.mq.jmqi.local.LocalMQ.loadLib(LocalMQ.java:1268)
    at com.ibm.mq.jmqi.local.LocalMQ$1.run(LocalMQ.java:309)
    at java.security.AccessController.doPrivileged(AccessController.java:400)
    at com.ibm.mq.jmqi.local.LocalMQ.initialise_inner(LocalMQ.java:259)
    at com.ibm.mq.jmqi.local.LocalMQ.initialise(LocalMQ.java:221)
    at com.ibm.mq.jmqi.local.LocalMQ.<init>(LocalMQ.java:1350)
    at com.ibm.mq.jmqi.local.LocalServer.<init>(LocalServer.java:230)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:86)

    at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.jav
a:58)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:542)
    at com.ibm.mq.jmqi.JmqiEnvironment.getInstance(JmqiEnvironment.java:706)
    at com.ibm.mq.jmqi.JmqiEnvironment.getMQI(JmqiEnvironment.java:640)
    at
com.ibm.msg.client.wmq.factories.WMQConnectionFactory.createV7ProviderConnection(WMQConnectionF
actory.java:8437)
    ... 7 more
Caused by: java.lang.UnsatisfiedLinkError: mqjbnd (Not found in java.library.path)
    at java.lang.ClassLoader.loadLibraryWithPath(ClassLoader.java:1235)
    at java.lang.ClassLoader.loadLibraryWithClassLoader(ClassLoader.java:1205)
    at java.lang.System.loadLibrary(System.java:534)
    at com.ibm.mq.jmqi.local.LocalMQ.loadLib(LocalMQ.java:1240)
    ... 20 more
```

2. After either the 32-bit or 64-bit environment has been set up, start the IBM MQ classes for JMS application using the command:

   ```
   java application-name
   ```

   where *application-name* is the name of the IBM MQ classes for JMS application to be run.

   An exception containing IBM MQ Reason code 2495 (MQRC_MODULE_NOT_FOUND) is thrown by the IBM MQ classes for JMS if:

   - The IBM MQ classes for JMS application is run in a 32-bit Java runtime environment, and a 64-bit environment has been set up for the IBM MQ classes for JMS, as the 32-bit Java runtime environment is unable to load the 64-bit Java Native Library.
   - The IBM MQ classes for JMS application is run in a 64-bit Java runtime environment, and a 32-bit environment has been set up for the IBM MQ classes for JMS, as the 64-bit Java runtime environment is unable to load the 32-bit Java Native Library.

The IBM MQ classes for JMS/Jakarta Messaging configuration file
An IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging configuration files specify properties that are used to configure IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

Note: The properties defined in configuration file can also be set as JVM system properties. If a property is set both in the configuration file and as a system property, the system property takes precedence. Therefore, if required, you can override any property in the configuration file by specifying it as a system property on the **java** command.

The format of an IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file is that of a standard Java properties file. A sample configuration file called `jms.config` is supplied in

the `bin` subdirectory of the IBM MQ classes for JMS installation directory. This file documents all the supported properties and their default values.

You can choose the name and location of an IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file. When you start your application, use a **java** command with the following format:

```
java -Dcom.ibm.msg.client.config.location= config_file_url application_name
```

In the command, *config_file_url* is a uniform resource locator (URL) that specifies the name and location of the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file. URLs of the following types are supported: http, file, ftp, and jar.

Here is an example of a **java** command:

```
java -Dcom.ibm.msg.client.config.location=file:/D:/mydir/myjms.config MyAppClass
```

This command identifies the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file as the file `D:\mydir\mjms.config` on the local Windows system.

When an application starts, IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging reads the contents of the configuration file and stores the specified properties in an internal property store. If the **java** command does not identify a configuration file, or if the configuration file cannot be found, IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging uses the default values for all the properties.

An IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file can be used with any of the supported transports between an application and a queue manager or broker.

Overriding properties specified in an IBM MQ MQI client configuration file

An IBM MQ MQI client configuration file can also specify properties that are used to configure IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging. However, properties specified in an IBM MQ MQI client configuration file apply only when an application connects to a queue manager in client mode.

If required, you can override any attribute in a IBM MQ MQI client configuration file by specifying it as a property in a IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file. To override an attribute in a IBM MQ MQI client configuration file, use an entry with the following format in the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file:

```
com.ibm.mq.cfg. stanza. propName = propValue
```

The variables in the entry have the following meanings:

stanza
The name of the stanza in the IBM MQ MQI client configuration file that contains the attribute

propName
The name of the attribute as specified in the IBM MQ MQI client configuration file

propValue
The value of the property that overrides the value of the attribute specified in the IBM MQ MQI client configuration file

Alternatively, you can override an attribute in an IBM MQ MQI client configuration file by specifying the property as a system property on the **java** command. Use the preceding format to specify the property as a system property.

Only the following attributes in an IBM MQ MQI client configuration file are relevant to IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging. If you specify or override other attributes, it has no effect. Specifically, please note that the `ChannelDefinitionFile` and `ChannelDefinitionDirectory` in the CHANNELS stanza of the client configuration file are not used. See “Using a client channel definition

table with IBM MQ classes for JMS" on page 273 for details of how to use the CCDT with the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging.

Table 10. Which stanza of the client configuration file contains which attribute	
Stanza	**Attribute**
CHANNELS stanza of the client configuration file	Put1DefaultAlwaysSync
CHANNELS stanza of the client configuration file	DefRecon
CHANNELS stanza of the client configuration file	ReconDelay
CHANNELS stanza of the client configuration file	PasswordProtection
ClientExitPath stanza of the client configuration file	ExitsDefaultPath
ClientExitPath stanza of the client configuration file	ExitsDefaultPath64
ClientExitPath stanza of the client configuration file	JavaExitsClasspath
JMQI stanza of the client configuration file	useMQCSPauthentication
MessageBuffer stanza of the client configuration file	MaximumSize
MessageBuffer stanza of the client configuration file	PurgeTime
MessageBuffer stanza of the client configuration file	UpdatePercentage
TCP stanza of the client configuration file	ClntRcvBufSize
TCP stanza of the client configuration file	ClntSndBufSize
TCP stanza of the client configuration file	Connect_Timeout
TCP stanza of the client configuration file	KeepAlive

For further details on the IBM MQ MQI client configuration see IBM MQ MQI client configuration file, `mqclient.ini`

Using Java Standard Environment Trace to configure JMS trace
Use the Java Standard Environment Trace Settings stanza to configure the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging trace facility.

`com.ibm.msg.client.commonservices.trace.outputName` = *traceOutputName*
`traceOutputName` is the directory and file name to which trace output is sent.

By default, trace information is written to a trace file in the current working directory of the application. The name of the trace file depends upon the environment that the application is running in:

- JMS 3.0 From IBM MQ 9.3.0, if the application has loaded the IBM MQ classes for Jakarta Messaging from the relocatable JAR file `com.ibm.mq.jakarta.client.jar` (Jakarta Messaging 3.0) or the IBM MQ classes for JMS from the relocatable JAR file `com.ibm.mq.allclient.jar` (JMS 2.0), trace is written to a file called `mqjavaclient_%PID%.cl%u.trc`.
- If the application has loaded the IBM MQ classes for JMS from the relocatable JAR file `com.ibm.mq.allclient.jar`, trace is written to a file called `mqjavaclient_%PID%.cl%u.trc`.
- If the application has loaded the IBM MQ classes for JMS from the JAR file `com.ibm.mqjms.jar`, trace is written to a file called `mqjava_%PID%.cl%u.trc`.

where *%PID%* is the process identifier of the application that is being traced, and *%u* is a unique number to differentiate files between threads running trace under different Java classloaders.

If a process ID is unavailable, a random number is generated and prefixed with the letter `f`. To include the process ID in a file name you specify, use the string `%PID%`.

If you specify an alternative directory, it must exist, and you must have write permission for this directory. If you do not have write permission, the trace output is written to `System.err`.

com.ibm.msg.client.commonservices.trace.include = *includeList*
includeList is a list of packages and classes that are traced, or the special values `ALL` or `NONE`.

Separate package or class names with a semicolon, `;`. *includeList* defaults to `ALL`, and traces all packages and classes in IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging.

Note: You can include a package but then exclude subpackages of that package. For example, if you include package `a.b` and exclude package `a.b.x`, the trace includes everything in `a.b.y` and `a.b.z`, but not `a.b.x` or `a.b.x.1`.

com.ibm.msg.client.commonservices.trace.exclude = *excludeList*
excludeList is a list of packages and classes that are not traced, or the special values `ALL` or `NONE`.

Separate package or class names with a semicolon, `;`. *excludeList* defaults to `NONE`, and therefore excludes no packages and classes in IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging from being traced.

Note: You can exclude a package but then include subpackages of that package. For example, if you exclude package `a.b` and include package `a.b.x`, the trace includes everything in `a.b.x` and `a.b.x.1`, but not `a.b.y` or `a.b.z`.

Any package or class that is specified, at the same level, as both included and excluded is included.

com.ibm.msg.client.commonservices.trace.maxBytes = *maxArrayBytes*
maxArrayBytes is the maximum number of bytes that are traced from any byte arrays.

If *maxArrayBytes* is set to a positive integer, it limits the number of bytes in a byte-array that are written out to the trace file. It truncates the byte array after writing *maxArrayBytes* out. Setting *maxArrayBytes* reduces the size of the resulting trace file, and reduces the effect of tracing on the performance of the application.

A value of `0` for this property means that none of the contents of any byte arrays are sent to the trace file.

The default value is `-1`, which removes any limit on the number of bytes in a byte array that are sent to the trace file.

com.ibm.msg.client.commonservices.trace.limit = *maxTraceBytes*
maxTraceBytes is the maximum number of bytes that are written to a trace output file.

maxTraceBytes works with *traceCycles*. If the number of bytes of trace written is near to the limit, the file is closed, and a new trace output file is started.

A value of `0` means that a trace output file has zero length. The default value is `-1`, which means that the amount of data to be written to a trace output file is unlimited.

com.ibm.msg.client.commonservices.trace.count = *traceCycles*
traceCycles is the number of trace output files to cycle through.

If the current trace output file reaches the limit specified by *maxTraceBytes*, the file is closed. Further trace output is written to the next trace output file in sequence. Each trace output file is distinguished by a numeric suffix appended to the file name. The current or most recent trace output file is `mqjms.trc.0`, the next most recent trace output file is `mqjms.trc.1`. Older trace files follow the same numbering pattern up to the limit.

The default value of *traceCycles* is 1. If *traceCycles* is 1, when the current trace output file reaches its maximum size, the file is closed and deleted. A new trace output file with the same name is started. Therefore, only one trace output file exists at a time.

com.ibm.msg.client.commonservices.trace.parameter = *traceParameters*
traceParameters controls whether method parameters and return values are included in the trace.

traceParameters defaults to TRUE. If *traceParameters* is set to FALSE, only method signatures are traced.

com.ibm.msg.client.commonservices.trace.startup = *startup*
There is an initialization phase of IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging during which resources are allocated. The main trace facility is initialized during the resource allocation phase.

If *startup* is set to TRUE, startup trace is used. Trace information is produced immediately and includes the setup of all components, including the trace facility itself. Startup trace information can be used to diagnose configuration problems. Startup trace information is always written to `System.err`.

startup defaults to FALSE.

startup is checked before initialization is complete. For this reason, only specify the property on the command line as a Java system property. Do not specify it in the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging configuration file.

com.ibm.msg.client.commonservices.trace.compress = *compressedTrace*
Set *compressedTrace* to TRUE to compress trace output.

The default value of *compressedTrace* is FALSE.

If *compressedTrace* is set to TRUE, trace output is compressed. The default trace output file name has the extension `.trz`. If compression is set to FALSE, the default value, the file has the extension `.trc` to indicate it is uncompressed. However if the file name for the trace output has been specified in *traceOutputName* that name is used instead; no suffix is applied to the file.

Compressed trace output is smaller than uncompressed. Because there is less I/O, it can be written out faster than uncompressed trace. Compressed tracing has less effect on the performance of IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging than uncompressed tracing.

If *maxTraceBytes* and *traceCycles* are set, multiple compressed trace files are created in place of multiple flat files.

If IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging ends in an uncontrolled manner, a compressed trace file might not be valid. For this reason, trace compression must only be used when IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging closes down in a controlled manner. Only use trace compression if the problems being investigated do not cause the JVM itself to stop unexpectedly. Do not use trace compression when diagnosing problems that can result in `System.Halt()` shutdowns or abnormal, uncontrolled JVM terminations.

com.ibm.msg.client.commonservices.trace.level = *traceLevel*
traceLevel specifies a filtering level for the trace. The defined trace levels are as follows:

- `TRACE_NONE`: `0`
- `TRACE_EXCEPTION`: `1`
- `TRACE_WARNING`: `3`
- `TRACE_INFO`: `6`
- `TRACE_ENTRYEXIT`: `8`
- `TRACE_DATA`: `9`
- `TRACE_ALL`: `Integer.MAX_VALUE`

Each trace level includes all lower levels. For example, if trace level is set at `TRACE_INFO`, then any trace point with a defined level of `TRACE_EXCEPTION`, `TRACE_WARNING`, or `TRACE_INFO` is written to the trace. All other trace points are excluded.

com.ibm.msg.client.commonservices.trace.standalone = *standaloneTrace*

standaloneTrace controls whether the IBM MQ JMS client tracing service is used in a WebSphere Application Server environment.

If *standaloneTrace* is set to TRUE, the IBM MQ JMS client tracing properties are used to determine the trace configuration.

If *standaloneTrace* is set to FALSE, and the IBM MQ JMS client is running in an WebSphere Application Server container, the WebSphere Application Server trace service is used. The trace information that is generated depends upon the trace settings of the application server.

The default value of *standaloneTrace* is FALSE.

Logging stanza
Use the Logging stanza to configure the IBM MQ classes for JMS log facility.

The following properties can be included in the Logging stanza:

com.ibm.msg.client.commonservices.log.outputName = path
The name of the log file that is used by the IBM MQ classes for JMS log facility. The default value is `mqjms.log`, which is written to the current working directory for the Java Runtime Environment that the IBM MQ classes for JMS are running in.

The property can take one of the following values:

- a single path name
- a comma-separated list of path names (all data is logged to all files)

Each path name can be an absolute or relative path name or:

"stderr" or "System.err"
Represents the standard error stream.

"stdout" or "System.out"
Represents the standard output stream.

com.ibm.msg.client.commonservices.log.maxBytes
The maximum number of bytes that are logged from any call to log message data.

Positive integer
Data is written up to that value of bytes per log call.

0
No data is written.

-1
Unlimited data is written (default).

com.ibm.msg.client.commonservices.log.limit
The maximum number of bytes that are written to any 1 log file (default is 262144).

Positive integer
Data is written up to that value of bytes per log file.

0
No data is written.

-1
Unlimited data is written.

com.ibm.msg.client.commonservices.log.count
The number of log files to cycle through. As each file reaches `com.ibm.msg.client.commonservices.trace.limit` trace will begin in the next file, the default is 3.

Positive integer
Number of files to cycle through.

0
A single file.

Java SE Specifics stanza
Use the Java SE Specifics stanza to configure properties that are used when the IBM MQ classes for JMS are being used in a Java Standard Edition environment.

com.ibm.msg.client.commonservices.j2se.produceJavaCore = TRUE|FALSE
Determines whether a JavaCore file is written immediately after the IBM MQ classes for JMS has generated an FDC file. If this is set to TRUE a JavaCore file is produced in the working directory of the Java Runtime Environment in which the IBM MQ classes for JMS are running.

TRUE
Generate JavaCore, subject to the Java Runtime Environment's ability to do so.

FALSE
Do not generate JavaCore; this is the default value.

IBM MQ Properties stanza
Use the IBM MQ Properties stanza to set properties that affect how the IBM MQ classes for JMS interact with IBM MQ.

com.ibm.msg.client.wmq.compat.base.internal.MQQueue.smallMsgsBufferReductionThreshold
When an application that uses the IBM MQ classes for JMS is connecting to an IBM MQ queue manager using IBM MQ messaging provider migration mode, the IBM MQ classes for JMS uses a default buffer size of 4 KB when it is receives messages. If the message that the application is trying to get is larger than 4 KB, the IBM MQ classes for JMS resizes the buffer to be large enough to accommodate the message. The larger buffer size is then used when subsequent messages are received.

This property controls when the buffer size is reduced back to 4 KB. By default, when ten consecutive messages that are less than the larger buffer size are received, the buffer size is reduced back to 4 KB. To reset the buffer size back to 4 KB every time a message is received, set the property to the value 0.

0

The buffer always resets to the default size.

10

This is the default value. The buffer will be resized after the tenth message.

com.ibm.msg.client.wmq.receiveConversionCCSID
When an application that is using the IBM MQ classes for JMS is connecting to an IBM MQ queue manager using IBM MQ messaging provider normal mode, the `receiveConversionCCSID` property can be set to override the default CCSID value in the MQMD structure that is used to receive messages from the queue manager. By default, the MQMD contains a CCSID field set to 1208, but this can be changed if, for example, the queue manager is unable to convert messages to this code page.

Valid values are any valid CCSID number or one of the following values:

-1
Use the platform default.

1208
This is the default value.

Client-mode specifics stanza
Use the Client-mode specifics stanza to specify properties that are used when the IBM MQ classes for JMS connect to a queue manager that is using the CLIENT transport.

com.ibm.mq.polling.RemoteRequestEntry
Specifies the polling interval that the IBM MQ classes for JMS uses to check for broken connections when it is waiting for a response from a queue manager.

Positive integer
The number of milliseconds to wait before checking. The default value is 10000 or 10 seconds. The minimum value is 3000, and lower values are treated in the same way as this minimum value.

Properties used to configure JMS client behavior
Use these properties to configure the behavior of the JMS client.

com.ibm.mq.jms.SupportMQExtensions TRUE|FALSE
The JMS 2.0 specification introduces changes to the way certain behaviors work. IBM MQ 8.0 includes the property `com.ibm.mq.jms.SupportMQExtensions`, which can be set to *TRUE*, to revert these changed behaviors back to previous implementations. Reverting the changed behaviors might be necessary some for JMS 2.0 applications, and also for some applications that use the JMS 1.1 API but run against the IBM MQ 8.0 IBM MQ classes for JMS.

TRUE
The following three areas of functionality are reverted by setting SupportMQExtensions to TRUE:

Message priority
Messages can be assigned a priority, *0* - *9*. Before JMS 2.0, messages could also use the value *-1*, indicating that a queue's default priority is used. JMS 2.0 does not allow a message priority of *-1* to be set. Turning on `SupportMQExtensions` allows the value of *-1* to be used.

Client id
The JMS 2.0 specification requires that non-null client ids are checked for uniqueness when they make a connection. Turning on `SupportMQExtensions`, means that this requirement is disregarded, and that a client ID can be reused.

NoLocal
The JMS 2.0 specification requires that when this constant is turned on, a consumer cannot receive messages that are published by the same client ID. Before JMS 2.0, this attribute was set on a subscriber to prevent it receiving messages that are published by its own connection. Turning on `SupportMQExtensions` reverts this behavior to its previous implementation.

FALSE
The changes of behavior are retained.

com.ibm.msg.client.jms.ByteStreamReadOnlyAfterSend= TRUE|FALSE
From IBM MQ 8.0.0 Fix Pack 2, after an application has sent a Bytes or Stream message, IBM MQ classes for JMS can set the state of the message that has just been sent to either read only, or write only.

TRUE
The objects are set to read only after being sent. Setting this value maintains compatibility with the JMS 2.0 specification

FALSE
The objects are set to write only after being sent. This is the default value.

Related concepts
“SupportMQExtensions property” on page 316
The JMS 2.0 specification introduced changes to the way certain behaviors work. IBM MQ 8.0 and later includes the property **com.ibm.mq.jms.SupportMQExtensions**, which can be set to TRUE to revert these changed behaviors back to previous implementations.

STEPLIB configuration for IBM MQ classes for JMS on z/OS
On z/OS, the STEPLIB used at run time must contain the IBM MQ SCSQAUTH and SCSQANLE libraries. Specify these libraries in the startup JCL or using the `.profile` file.

From z/OS UNIX System Services, you can add these using a line in your `.profile` as shown in the following code snippet, replacing `thlqual` with the high-level data set qualifier that you chose when installing IBM MQ:

```
export STEPLIB=thlqual.SCSQAUTH:thlqual.SCSQANLE:$STEPLIB
```

In other environments, you typically need to edit the startup JCL to include SCSQAUTH and SCSQANLE on the STEPLIB concatenation:

```
STEPLIB DD DSN=thlqual.SCSQAUTH,DISP=SHR
        DD DSN=thlqual.SCSQANLE,DISP=SHR
```

IBM MQ classes for JMS and software management tools
Software management tools such as Apache Maven can be used with the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

Many large development organizations use these tools to centrally manage repositories of third-party libraries.

The IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are composed of a number of JAR files. When you are developing Java language applications by using this API, an installation of either an IBM MQ Server, IBM MQ Client, or IBM MQ Client SupportPac is required on the machine where the application is being developed.

If you want to use such a tool and add the JAR files that make up the IBM MQ classes for JMS to a centrally managed repository, the following points must be observed:

- A repository or container must be made available only to developers within your organization. Any distribution outside of the organization is not permitted.
- The repository needs to contain a complete and consistent set of JAR files from a single IBM MQ release or Fix Pack.
- You are responsible for updating the repository with any maintenance provided by IBM Support.

The following JAR files need to be installed into the repository:

- JMS 2.0 `com.ibm.mq.allclient.jar` and `jms.jar` are required if you are using the IBM MQ classes for JMS.
- JMS 3.0 `com.ibm.mq.jakarta.client.jar` and `jakarta.jms-api.jar` are required if you are using IBM MQ classes for Jakarta Messaging.
- `fscontext.jar` is required if you are using IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging and are accessing JMS administered objects that are stored in a file system JNDI context.
- `providerutil.jar` is required if you are using the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging and are accessing JMS administered objects that are stored in a file system JNDI context.
- The Bouncy Castle security provider and CMS support JAR files are required for support for non-IBM JREs. For more information, see Support for non-IBM JREs.

Running IBM MQ classes for JMS applications under the Java security manager
IBM MQ classes for JMS can run with the Java security manager enabled. To run applications successfully with the Java security manager enabled, you must configure your Java Virtual Machine (JVM) with a suitable policy configuration file.

The simplest way to create a suitable policy definition file is to change the policy configuration file supplied with your Java runtime environment (JRE). On most systems, this file is in the directory `lib/security/java.policy` relative to your JRE directory. You can edit the policy configuration file either by using your preferred editor or by using the policy tool program supplied with your JRE.

Example policy configuration file

Here is an example of a policy configuration file that allows IBM MQ classes for JMS to run successfully under the default security manager. This file will need to be customized, to specify the locations of certain files and directories: *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed, *MQ_DATA_DIRECTORY* represents the location of the MQ data directory, and *QM_NAME* is the name of the queue manager for which access is being configured.

```
grant codeBase "file:MQ_INSTALLATION_PATH/java/lib/*" {
```

```
  //We need access to these properties, mainly for tracing
  permission java.util.PropertyPermission "user.name","read";
  permission java.util.PropertyPermission "os.name","read";
  permission java.util.PropertyPermission "user.dir","read";
  permission java.util.PropertyPermission "line.separator","read";
  permission java.util.PropertyPermission "path.separator","read";
  permission java.util.PropertyPermission "file.separator","read";
  permission java.util.PropertyPermission "com.ibm.msg.client.commonservices.log.*","read";
  permission java.util.PropertyPermission "com.ibm.msg.client.commonservices.trace.*","read";
  permission java.util.PropertyPermission "Diagnostics.Java.Errors.Destination.Filename","read";
  permission java.util.PropertyPermission "com.ibm.mq.commonservices","read";
  permission java.util.PropertyPermission "com.ibm.mq.cfg.*","read";

  //Tracing - we need the ability to control java.util.logging
  permission java.util.logging.LoggingPermission "control";
  // And access to create the trace file and read the log file - assumed to be in the current
directory
  permission java.io.FilePermission "*","read,write";

  // We'd like to set up an mBean to control trace
  permission javax.management.MBeanServerPermission "createMBeanServer";
  permission javax.management.MBeanPermission "*","*";

  // We need to be able to read manifests etc from the jar files in the installation directory
  permission java.io.FilePermission "MQ_INSTALLATION_PATH/java/lib/-","read";

  //Required if mqclient.ini/mqs.ini configuration files are used
  permission java.io.FilePermission "MQ_DATA_DIRECTORY/mqclient.ini","read";
  permission java.io.FilePermission "MQ_DATA_DIRECTORY/mqs.ini","read";

  //For the client transport type.
  permission java.net.SocketPermission "*","connect,resolve";

  //For the bindings transport type.
  permission java.lang.RuntimePermission "loadLibrary.*";

  //For applications that use CCDT tables (access to the CCDT AMQCLCHL.TAB)
  permission java.io.FilePermission "MQ_DATA_DIRECTORY/qmgrs/QM_NAME/@ipcc/AMQCLCHL.TAB","read";

  //For applications that use User Exits
  permission java.io.FilePermission "MQ_DATA_DIRECTORY/exits/*","read";
  permission java.io.FilePermission "MQ_DATA_DIRECTORY/exits64/*","read";
  permission java.lang.RuntimePermission "createClassLoader";

  //Required for the z/OS platform
  permission java.util.PropertyPermission "com.ibm.vm.bitmode","read";

  // Used by the internal ConnectionFactory implementation
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";

  // Used for controlled class loading
  permission java.lang.RuntimePermission "setContextClassLoader";

  // Used to default the Application name in Client mode connections
  permission java.util.PropertyPermission "sun.java.command","read";

  // Used by the IBM JSSE classes
  permission java.util.PropertyPermission "com.ibm.crypto.provider.AESNITrace","read";

  //Required to determine if an IBM Java Runtime is running in FIPS mode,
  //and to modify the property values status as required.
  permission java.util.PropertyPermission "com.ibm.jsse2.usefipsprovider","read,write";
  permission java.util.PropertyPermission "com.ibm.jsse2.JSSEFIPS","read,write";
  //Required if an IBM FIPS provider is to be used for SSL communication.
  permission java.security.SecurityPermission "insertProvider.IBMJCEFIPS";

  // Required for non-IBM Java Runtimes that establish secure client
  // transport mode connections using mutual TLS authentication
  permission java.util.PropertyPermission "javax.net.ssl.keyStore","read";
  permission java.util.PropertyPermission "javax.net.ssl.keyStorePassword","read";
};
```

In the example, the `grant` statement contains the permissions required by IBM MQ classes for JMS. To use these grant statements in your policy configuration file, you might need to modify the path names depending on where you have installed IBM MQ classes for JMS and where you store your applications.

The sample applications supplied with IBM MQ classes for JMS, and scripts to run them, do not enable the security manager.

Important:

The IBM MQ classes for JMS trace facility requires further permissions as it performs additional querying of system properties, and also further file system operations.

A suitable template security policy file for running under a security manager with tracing enabled is provided in the `samples/wmqjava` directory of the IBM MQ installation as `example.security.policy`.

Post installation setup for IBM MQ classes for JMS applications

This topic tells you what authorities IBM MQ classes for JMS applications need in order to access the resources of a queue manager. It also introduces connection modes and describes how to configure a queue manager so that applications can connect in client mode.

Remember to check the IBM MQ readme file. It might contain information that supersedes the information in this topic.

Objects used by JMS that require authorization for non-privileged users

Non-privileged users need authorization granted to access the queues used by JMS. Every JMS application needs authorization to the queue manager with which it works.

For details about access control in IBM MQ, see Setting up security.

IBM MQ classes for JMS applications need `connect` and `inq` authority to the queue manager. You can set appropriate authorizations using the **`setmqaut`** control command, for example:

```
setmqaut -m QM1 -t qmgr -g jmsappsgroup +connect +inq
```

For the point-to-point domain, the following authorities are required:

- Queues that are used by MessageProducer objects need `put` authority.
- Queues that are used by MessageConsumer and QueueBrowser objects need `get`, `inq`, and `browse` authorities.
- The QueueSession.createTemporaryQueue() method needs access to the model queue specified by the TEMPMODEL property of the QueueConnectionFactory object. By default this model queue is SYSTEM.TEMP.MODEL.QUEUE.

If any of these queues are alias queues, their target queues require inquire authority. If the target queue is a cluster queue it also requires browse authority.

For the publish/subscribe domain, the following queues are used if the IBM MQ classes for JMS are connecting to an IBM MQ queue manager in IBM MQ messaging provider migration mode:

- SYSTEM.JMS.ADMIN.QUEUE
- SYSTEM.JMS.REPORT.QUEUE
- SYSTEM.JMS.MODEL.QUEUE
- SYSTEM.JMS.PS.STATUS.QUEUE
- SYSTEM.JMS.ND.SUBSCRIBER.QUEUE
- SYSTEM.JMS.D.SUBSCRIBER.QUEUE
- SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE
- SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE
- SYSTEM.BROKER.CONTROL.QUEUE

For further information on IBM MQ messaging provider migration mode, see Configuring the JMS **`PROVIDERVERSION`** property

Additionally, if the IBM MQ classes for JMS are connecting to a queue manager in this mode, any application that publishes messages needs access to the stream queue specified by the TopicConnectionFactory or topic object. By default, this queue is SYSTEM.BROKER.DEFAULT.STREAM.

If you use ConnectionConsumer, IBM MQ Resource Adapter, or the WebSphere Application Server IBM MQ messaging provider, additional authorization might be needed.

Queues to be read by the ConnectionConsumer must have `get`, `inq`, and `browse` authorities. The system dead-letter queue, and any backout-requeue queue or report queue used by the ConnectionConsumer must have `put` and `passall` authorities.

When an application uses IBM MQ messaging provider normal mode to perform publish/subscribe messaging, the application makes use of the integrated publish/subscribe functionality provided by the queue manager. See Publish/subscribe security for information on securing the topics and queues that are used.

Connection modes for IBM MQ classes for JMS
An IBM MQ classes for JMS application can connect to a queue manager in either client or bindings mode. In client mode, IBM MQ classes for JMS connects to the queue manager over TCP/IP. In bindings mode, IBM MQ classes for JMS connects directly to the queue manager using the Java Native Interface (JNI).

On z/OS, bindings mode can be used in any environment, but client mode can be used only in the following environments:

- In WebSphere Application Server or WebSphere Liberty Profile connecting to any queue manager, on any platform, including z/OS.
- In batch environments when connecting to an IBM MQ for z/OS queue manager, running on any LPAR.

An application running on any other platform can connect to a queue manager in either bindings or client mode.

You can use the current or any earlier supported version of IBM MQ classes for JMS with a current queue manager, and you can use a current or earlier supported version of queue manager with the current version of IBM MQ classes for JMS. If you mix different versions, function is limited to the level of the earlier version.

The following sections describe each of the connection modes in more detail.

Client mode

To connect to a queue manager in client mode, an IBM MQ classes for JMS application can run on the same system on which the queue manager is running, or on a different system. In each case, IBM MQ classes for JMS connects to the queue manager over TCP/IP.

Bindings mode

To connect to a queue manager in bindings mode, an IBM MQ classes for JMS application must run on the same system on which the queue manager is running.

The IBM MQ classes for JMS connects directly to the queue manager using the Java Native Interface (JNI). To use the bindings transport, the IBM MQ classes for JMS must be run in an environment that has access to the IBM MQ Java Native Interface libraries; see “Configuring the Java Native Interface (JNI) libraries” on page 92 for further information.

The IBM MQ classes for JMS support the following values for *ConnectOption*:

- MQCNO_FASTPATH_BINDING
- MQCNO_STANDARD_BINDING
- MQCNO_SHARED_BINDING
- MQCNO_ISOLATED_BINDING
- MQCNO_RESTRICT_CONN_TAG_QSG
- MQCNO_RESTRICT_CONN_TAG_Q_MGR

To change the connection options used by the IBM MQ classes for JMS, modify the Connection Factory property CONNOPT.

For further information on connection options, see “Connecting to a queue manager using the MQCONNX call” on page 712

To use the bindings transport, the Java Runtime Environment being used must support the Coded Character Set Identifier (CCSID) of the queue manager that the IBM MQ classes for JMS are connecting to.

Details on how to determine what CCSIDs are supported by a Java Runtime Environment can be found in IBM MQ FDC with Probe ID 21 generated when using the IBM MQ V7 classes for Java or IBM MQ V7 classes for JMS.

Configuring your queue manager so that IBM MQ classes for JMS applications can connect in client mode
To configure your queue manager so that IBM MQ classes for JMS applications can connect in client mode, you must create a server connection channel definition and start a listener.

Creating a server connection channel definition

On all platforms, you can use the MQSC command DEFINE CHANNEL to create a server connection channel definition. See the following example:

```
DEFINE CHANNEL(JAVA.CHANNEL) CHLTYPE(SVRCONN) TRPTYPE(TCP)
```

IBM i On IBM i, you can use the CL command CRTMQMCHL instead, as in the following example:

```
CRTMQMCHL CHLNAME(JAVA.CHANNEL) CHLTYPE(*SVRCN)
TRPTYPE(*TCP)
MQMNAME(QMGRNAME)
```

In this command, *QMGRNAME* is the name of your queue manager.

Windows Linux On Linux and Windows, you can also create a server connection channel definition using IBM MQ Explorer.

z/OS On z/OS you can use the operations and control panels to create a server connection channel definition .

The name of the channel (JAVA.CHANNEL in the previous examples) must be the same as the channel name specified by the CHANNEL property of the connection factory that your application uses to connect to the queue manager. The default value of the CHANNEL property is SYSTEM.DEF.SVRCONN.

Starting a listener

You must start a listener for your queue manager if one is not already started.

Multi On Multiplatforms, you can use the MQSC command START LISTENER to start a listener after first creating a listener object by using the MQSC command DEFINE LISTENER, as shown in the following example:

```
DEFINE LISTENER(LISTENER.TCP) TRPTYPE(TCP) PORT(1414)
START LISTENER(LISTENER.TCP)
```

z/OS On z/OS, you use only the START LISTENER command, as in the following example, but note that the channel initiator address space must be started before you can start a listener:

```
START LISTENER TRPTYPE(TCP) PORT(1414)
```

IBM i On IBM i, you can also use the CL command STRMQMLSR to start a listener, as in the following example:

```
STRMQMLSR PORT(1414) MQMNAME(QMGRNAME)
```

In this command, *QMGRNAME* is the name of your queue manager.

ALW On AIX, Linux, and Windows, you can also use the control command **runmqlsr** to start a listener, as in the following example:

```
runmqlsr -t tcp -p 1414 -m QMgrName
```

In this command, *QMgrName* is the name of your queue manager.

Windows Linux On Linux and Windows, you can also start a listener using IBM MQ Explorer.

z/OS On z/OS, you can also use the operations and control panels to start a listener.

The number of the port on which the listener is listening must be the same as the port number specified by the PORT property of the connection factory that your application uses to connect to the queue manager. The default value of the PORT property is 1414.

The point-to-point IVT for IBM MQ classes for JMS

A point-to-point installation verification test (IVT) program is supplied with IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging. The program connects to a queue manager in either bindings or client mode, sends a message to the queue called SYSTEM.DEFAULT.LOCAL.QUEUE, and then receives the message from the queue. The program can create and configure all the objects that it requires dynamically at run time, or it can use JNDI to retrieve administered objects from a directory service.

Run the installation verification test without using JNDI first because the test is self contained and does not require the use of a directory service. For a description of administered objects, see Configuring JMS objects using the administration tool.

The point-to-point installation verification test without using JNDI

In this test, the IVT program creates and configures all the objects that it requires dynamically at run time and does not use JNDI.

Multi On multiplatforms, a script is provided to run the IVT program. The script is called **IVTRun** on AIX and Linux systems and **IVTRun.bat** on Windows. the script is located in the `bin` subdirectory of the IBM MQ classes for JMS installation directory. The classpath must contain `com.ibm.mqjms.jar`.

To run the test in bindings mode, enter the following command:

```
IVTRun -nojndi [-m qmgr ] [-v providerVersion ] [-t]
```

To run the test in client mode, first set up the queue manager as described in “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028. Note that the channel to be used defaults to **SYSTEM.DEF.SVRCONN** and the queue to be used is **SYSTEM.DEFAULT.LOCAL.QUEUE**, then enter the following command:

```
IVTRun -nojndi -client -m qmgr -host hostname [-port port ] [-channel channel ]
[-v providerVersion ] [-ccsid ccsid ] [-t]
```

z/OS No equivalent script is provided on z/OS systems. Instead, you run the IVT in bindings mode by invoking the Java class directly. On z/OS, you choose between two functionally identical instances of the IVT program:

- `com.ibm.mq.jms.MQJMSIVT`, which is available with IBM MQ classes for JMS (JMS 2.0). To use this program, the classpath must contain `com.ibm.mqjms.jar` or `com.ibm.mq.allclient.jar`.
- `com.ibm.mq.jakarta.jms.MQJMSIVT`, which is available with IBM MQ classes for Jakarta Messaging (Jakarta Messaging 3.0). To use this program, the classpath must contain `com.ibm.mq.jakarta.client.jar`.

To run the test in bindings mode on z/OS, enter the following command:

```
java com.ibm.mq.jms.MQJMSIVT -nojndi [-m qmgr ] [-v providerVersion ] [-t]
```

The parameters on the commands have the following meanings:

-m *qmgr*
The name of the queue manager to which the IVT program connects. If you run the test in bindings mode and omit this parameter, the IVT program connects to the default queue manager.

-host *hostname*
The host name or IP address of the system on which the queue manager is running.

-port *port*
The number of the port on which the listener of the queue manager is listening. The default value is 1414.

-channel *channel*
The name of the MQI channel that the IVT program uses to connect to the queue manager. The default value is SYSTEM.DEF.SVRCONN.

-v *providerVersion*
The release level of the queue manager to which the IVT program expects to connect.

This parameter is used to set the PROVIDERVERSION property of an MQQueueConnectionFactory object and has the same valid values as those of the PROVIDERVERSION property. For more information about this parameter therefore, including its valid values, see JMS: changes to PROVIDERVERSION property and the description of the PROVIDERVERSION property in Properties of IBM MQ classes for JMS objects.

The default value is `unspecified`.

-ccsid *ccsid*
The identifier (CCSID) of the coded character set, or code page, to be used by the connection. The default value is 819.

-t
Tracing is enabled. By default, tracing is disabled.

A successful test produces output similar to the following sample output:

```
5724-H72, 5655-R36, 5724-L26, 5655-L82 (c) Copyright IBM Corp. 2008, 2024. All
Rights Reserved.
WebSphere MQ classes for Java(tm) Message Service 7.0
Installation Verification Test

Creating a QueueConnectionFactory
Creating a Connection
Creating a Session
Creating a Queue
Creating a QueueSender
Creating a QueueReceiver
Creating a TextMessage
Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE
Reading the message back again

Got message
JMSMessage class: jms_text
JMSType:     null
JMSDeliveryMode: 2
JMSExpiration:  0
JMSPriority:   4
```

```
JMSMessageID:   ID:414d5120514d5f6d627720202020202001edb14620005e03
JMSTimestamp:   1187170264000
JMSCorrelationID: null
JMSDestination:  queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
JMSReplyTo:    null
JMSRedelivered:  false
JMSXUserID: mwhite
JMS_IBM_Encoding: 273
JMS_IBM_PutApplType: 28
JMSXAppID: IBM MQ Client for Java
JMSXDeliveryCount: 1
JMS_IBM_PutDate: 20070815
JMS_IBM_PutTime: 09310400
JMS_IBM_Format: MQSTR
JMS_IBM_MsgType: 8
A simple text message from the MQJMSIVT
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
```

The point-to-point installation verification test using JNDI

In this test, the IVT program uses JNDI to retrieve administered objects from a directory service.

Before you can run the test, you must configure a directory service that is based on a Lightweight Directory Access Protocol (LDAP) server or the local file system. You must also configure the IBM MQ JMS administration tool so that it can use the directory service to store administered objects. For more information about these prerequisites, see “Prerequisites for IBM MQ classes for JMS” on page 85. For information about how to configure the IBM MQ JMS administration tool, see Configuring the JMS administration tool.

The IVT program must be able to use JNDI to retrieve an MQQueueConnectionFactory object and an MQQueue object from the directory service. A script is provided to create these administered objects for you. The script is called IVTSetup on AIX and Linux systems and IVTSetup.bat on Windows, and is in the `bin` subdirectory of the IBM MQ classes for JMS installation directory. To run the script, enter the following command:

```
IVTSetup
```

The script invokes the IBM MQ JMS administration tool to create the administered objects.

The MQQueueConnectionFactory object is bound with the name ivtQCF and is created with the default values for all its properties, which means that the IVT program runs in bindings mode and connects to the default queue manager. If you want the IVT program to run in client mode, or connect to a queue manager other than the default queue manager, you must use the IBM MQ JMS administration tool or IBM MQ Explorer to change the appropriate properties of the MQQueueConnectionFactory object. For information about how to use the IBM MQ Explorer JMS administration tool, see Configuring JMS objects using the administration tool. For information about how to use IBM MQ Explorer, see Introduction to IBM MQ Explorer or the help provided with IBM MQ Explorer.

The MQQueue object is bound with the name ivtQ and is created with the default values for all its properties, except for the QUEUE property, which has the value SYSTEM.DEFAULT.LOCAL.QUEUE.

When you have created the administered objects, you can run the IVT program. To run the test using JNDI, enter the following command:

```
IVTRun -url "providerURL" [-icf initCtxFact ] [-t]
```

The parameters on the command have the following meanings:

-url "*providerURL*"
The uniform resource locator (URL) of the directory service. The URL can have one of the following formats:

- `ldap://`*`hostname`*`/`*`contextName`* , for a directory service based on an LDAP server
- `file:/`*`directoryPath`* , for a directory service based on the local file system

Note that you must enclose the URL in quotation marks (").

-icf *initCtxFact*
The class name of the initial context factory, which must be one of the following values:

- `com.sun.jndi.ldap.LdapCtxFactory`, for a directory service based on an LDAP server. This is the default value.
- `com.sun.jndi.fscontext.RefFSContextFactory`, for a directory service based on the local file system.

-t
Tracing is enabled. By default, tracing is disabled.

A successful test produces output similar to that for a successful test without using JNDI. The main difference is that the output indicates that the test is using JNDI to retrieve an MQQueueConnectionFactory object and an MQQueue object.

Although not strictly necessary, it is good practice to tidy up after the test by deleting the administered objects created by the IVTSetup script. A script is provided for this purpose. The script is called IVTTidy on AIX and Linux systems and IVTTidy.bat on Windows, and is in the `bin` subdirectory of the IBM MQ classes for JMS installation directory.

Problem determination for the point-to-point installation verification test

The installation verification test might fail for the following reasons:

- If the IVT program writes a message indicating that it cannot find a class, check that your class path is set correctly, as described in “Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging” on page 90.
- The test might fail with the following message:

```
Failed to connect to queue manager ' qmgr ' with connection mode ' connMode '
and host name ' hostname '
```

  and an associated reason code of 2059. The variables in the message have the following meanings:

  qmgr
  The name of the queue manager to which the IVT program is trying to connect. This message insert is blank if the IVT program is trying to connect to the default queue manager in bindings mode.

  connMode
  The connection mode, which is either `Bindings` or `Client`.

  hostname
  The host name or IP address of the system on which the queue manager is running.

  This message means that the queue manager to which the IVT program is trying to connect is not available. Check that the queue manager is running and, if the IVT program is trying to connect to the default queue manager, make sure that the queue manager is defined as the default queue manager for your system.
- The test might fail with the following message:

```
Failed to open MQ queue 'SYSTEM.DEFAULT.LOCAL.QUEUE'
```

This message means that the queue SYSTEM.DEFAULT.LOCAL.QUEUE does not exist on the queue manager to which the IVT program is connected. Alternatively, if the queue does exist, the IVT program cannot open the queue because it is not enabled for putting and getting messages. Check that the queue exists and that it is enabled for putting and getting messages.

- The test might fail with the following message:

```
Unable to bind to object
```

This message means that there is a connection to the LDAP server, but that the LDAP server is not correctly configured. Either the LDAP server is not configured for storing Java objects, or the permissions on the objects or the suffix are not correct. For more help in this situation, see the documentation for your LDAP server.

- The test might fail with the following message:

```
The security authentication was not valid that was supplied for
QueueManager ' qmgr ' with connection mode 'Client' and host name ' hostname '
```

This message means that the queue manager is not correctly set up to accept a client connection from your system. See “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028 for details.

The publish/subscribe IVT for IBM MQ classes for JMS

A publish/subscribe installation verification test (IVT) program is supplied with IBM MQ classes for JMS. The program connects to a queue manager in either bindings or client mode, subscribes to a topic, publishes a message on the topic, and then receives the message that it has just published. The program can create and configure all the objects that it requires dynamically at run time, or it can use JNDI to retrieve administered objects from a directory service.

Run the installation verification test without using JNDI first because the test is self contained and does not require the use of a directory service. For a description of administered objects, see Configuring JMS objects using the administration tool.

The publish/subscribe installation verification test without using JNDI

In this test, the IVT program creates and configures all the objects that it requires dynamically at run time and does not use JNDI.

A script is provided to run the IVT program. The script is called PSIVTRun on AIX and Linux systems and PSIVTRun.bat on Windows, and is in the `bin` subdirectory of the IBM MQ classes for JMS installation directory.

To run the test in bindings mode, enter the following command:

```
PSIVTRun -nojndi [-m qmgr ] [-bqm brokerQmgr ] [-v providerVersion ] [-t]
```

To run the test in client mode, first set up the queue manager as described in “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028 noting that the channel to be used defaults to SYSTEM.DEF.SVRCONN, then enter the following command:

```
PSIVTRun -nojndi -client -m qmgr -host hostname [-port port ] [-channel channel ]
[-bqm brokerQmgr ] [-v providerVersion ] [-ccsid ccsid ] [-t]
```

The parameters on the commands have the following meanings:

-m *qmgr*
The name of the queue manager to which the IVT program connects. If you run the test in bindings mode and omit this parameter, the IVT program connects to the default queue manager.

-host *hostname*
The host name or IP address of the system on which the queue manager is running.

-port *port*
The number of the port on which the listener of the queue manager is listening. The default value is 1414.

-channel *channel*
The name of the MQI channel that the IVT program uses to connect to the queue manager. The default value is `SYSTEM.DEF.SVRCONN`.

-bqm *brokerQmgr*
The name of the queue manager on which the broker is running. The default value is the name of the queue manager to which the IVT program connects.

This parameter is not relevant for queue manager version number v of 7, or greater.

-v *providerVersion*
The release level of the queue manager to which the IVT program expects to connect.

This parameter is used to set the PROVIDERVERSION property of an MQTopicConnectionFactory object and has the same valid values as those of the PROVIDERVERSION property. For more information about this parameter therefore, including its valid values, see the description of the PROVIDERVERSION property in Properties of IBM MQ classes for JMS objects.

The default value is `unspecified`.

-ccsid *ccsid*
The identifier (CCSID) of the coded character set, or code page, to be used by the connection. The default value is 819.

-t
Tracing is enabled. By default, tracing is disabled.

A successful test produces output similar to the following sample output:

```
5724-H72, 5655-R36, 5724-L26, 5655-L82 (c) Copyright IBM Corp. 2008, 2024. All
Rights Reserved.
IBM MQ classes for Java Message Service 7.0
Publish/Subscribe Installation Verification Test

Creating a TopicConnectionFactory
Creating a Connection
Creating a Session
Creating a Topic
Creating a TopicPublisher
Creating a TopicSubscriber
Creating a TextMessage
Adding text
Publishing the message to topic://MQJMS/PSIVT/Information
Waiting for a message to arrive [5 secs max]...

Got message:
JMSMessage class: jms_text
JMSType:     null
JMSDeliveryMode: 2
JMSExpiration:  0
JMSPriority:   4
JMSMessageID:   ID:414d5120514d5f6d627720202020202001edb14620006706
JMSTimestamp:   1187182520203
JMSCorrelationID: ID:414d5120514d5f6d627720202020202001edb14620006704
JMSDestination:  topic://MQJMS/PSIVT/Information
JMSReplyTo:    null
JMSRedelivered:  false
JMSXUserID: mwhite
JMS_IBM_Encoding: 273
```

```
JMS_IBM_PutApplType: 26
JMSXAppID: QM_mbw
JMSXDeliveryCount: 1
JMS_IBM_PutDate: 20070815
JMS_IBM_ConnectionID: 414D5143514D5F6D627720202020202001EDB14620006601
JMS_IBM_PutTime: 12552020
JMS_IBM_Format: MQSTR
JMS_IBM_MsgType: 8
A simple text message from the MQJMSPSIVT program
Reply string equals original string
Closing TopicSubscriber
Closing TopicPublisher
Closing Session
Closing Connection
PSIVT finished
```

The publish/subscribe installation verification test using JNDI

In this test, the IVT program uses JNDI to retrieve administered objects from a directory service.

Before you can run the test, you must configure a directory service that is based on a Lightweight Directory Access Protocol (LDAP) server or the local file system. You must also configure the IBM MQ JMS administration tool so that it can use the directory service to store administered objects. For more information about these prerequisites, see “Prerequisites for IBM MQ classes for JMS” on page 85. For information about how to configure the IBM MQ JMS administration tool, see Configuring the JMS administration tool.

The IVT program must be able to use JNDI to retrieve an MQTopicConnectionFactory object and an MQTopic object from the directory service. A script is provided to create these administered objects for you. The script is called IVTSetup on AIX and Linux systems and IVTSetup.bat on Windows, and is in the `bin` subdirectory of the IBM MQ classes for JMS installation directory. To run the script, enter the following command:

```
IVTSetup
```

The script invokes the IBM MQ JMS administration tool to create the administered objects.

The MQTopicConnectionFactory object is bound with the name ivtTCF and is created with the default values for all its properties, which means that the IVT program runs in bindings mode, connects to the default queue manager, and uses the embedded publish/subscribe function. If you want the IVT program to run in client mode, connect to a queue manager other than the default queue manager, or use IBM Integration Bus instead of the embedded publish/subscribe function, you must use the IBM MQ JMS administration tool or IBM MQ Explorer to change the appropriate properties of the MQTopicConnectionFactory object. For information about how to use the IBM MQ JMS administration tool, see Configuring JMS objects using the administration tool. For information about how to use IBM MQ Explorer, see the help provided with IBM MQ Explorer.

The MQTopic object is bound with the name ivtT and is created with the default values for all its properties, except for the TOPIC property, which has the value MQJMS/PSIVT/Information.

When you have created the administered objects, you can run the IVT program. To run the test using JNDI, enter the following command:

```
PSIVTRun -url "providerURL" [-icf initCtxFact ] [-t]
```

The parameters on the command have the following meanings:

-url "*providerURL*"
The uniform resource locator (URL) of the directory service. The URL can have one of the following formats:

- `ldap://`*`hostname`*`/`*`contextName`* , for a directory service based on an LDAP server
- `file:/`*`directoryPath`* , for a directory service based on the local file system

Note that you must enclose the URL in quotation marks (").

-icf *initCtxFact*
The class name of the initial context factory, which must be one of the following values:

- `com.sun.jndi.ldap.LdapCtxFactory`, for a directory service based on an LDAP server. This is the default value.
- `com.sun.jndi.fscontext.RefFSContextFactory`, for a directory service based on the local file system.

-t
Tracing is enabled. By default, tracing is disabled.

A successful test produces output similar to that for a successful test without using JNDI. The main difference is that the output indicates that the test is using JNDI to retrieve an MQTopicConnectionFactory object and an MQTopic object.

Although not strictly necessary, it is good practice to tidy up after the test by deleting the administered objects created by the IVTSetup script. A script is provided for this purpose. The script is called IVTTidy on AIX and Linux systems and IVTTidy.bat on Windows, and is in the `bin` subdirectory of the IBM MQ classes for JMS installation directory.

Problem determination for the publish/subscribe installation verification test

The installation verification test might fail for the following reasons:

- If the IVT program writes a message indicating that it cannot find a class, check that your class path is set correctly, as described in "Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90.
- The test might fail with the following message:

```
Failed to connect to queue manager ' qmgr ' with
connection mode ' connMode ' and host name ' hostname '
```

  and an associated reason code of 2059. The variables in the message have the following meanings:

  qmgr
  The name of the queue manager to which the IVT program is trying to connect. This message insert is blank if the IVT program is trying to connect to the default queue manager in bindings mode.

  connMode
  The connection mode, which is either `Bindings` or `Client`.

  hostname
  The host name or IP address of the system on which the queue manager is running.

  This message means that the queue manager to which the IVT program is trying to connect is not available. Check that the queue manager is running and, if the IVT program is trying to connect to the default queue manager, make sure that the queue manager is defined as the default queue manager for your system.
- The test might fail with the following message:

```
Unable to bind to object
```

  This message means that there is a connection to the LDAP server, but that the LDAP server is not correctly configured. Either the LDAP server is not configured for storing Java objects, or the permissions on the objects or the suffix are not correct. For more help in this situation, see the documentation for your LDAP server.
- The test might fail with the following message:

```
The security authentication was not valid that was supplied for
QueueManager 'qmgr' with connection mode 'Client' and host name 'hostname'
```

This message means that the queue manager is not set up correctly to accept a client connection from your system. For more information, see “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028.

JMS 2.0 *Using the IBM MQ classes for JMS sample applications*

The IBM MQ classes for JMS sample applications provide an overview of the common features of the JMS API. You can use them to verify your installation and messaging server set up and to help you build your own applications.

About this task

If you need help to create your own applications, you can use the sample applications as a starting point. Both the source and a compiled version are provided for each application. Review the sample source code and identify the key steps to create each required object for your application (ConnectionFactory, Connection, Session, Destination, and a Producer, or a Consumer, or both), and to set any specific properties that are needed to specify how you want your application to work. For more information, see “Writing IBM MQ classes for JMS/Jakarta Messaging applications” on page 134. The samples might be subject to change in future releases of IBM MQ.

For JMS 2.0, Table 11 on page 115 shows where the IBM MQ classes for JMS sample applications are installed on each platform.

Note:

JMS 3.0 For IBM MQ classes for Jakarta Messaging, new samples are being prepared.

Table 11. Installation directories for the IBM MQ classes for JMS sample applications

Platform	Directory
AIX AIX Linux Linux	*MQ_INSTALLATION_PATH*/samp/jms/samples
Windows Windows	*MQ_INSTALLATION_PATH*\tools\jms\samples
IBM i IBM i	/qibm/proddata/mqm/java/samples/jms/samples
z/OS z/OS	*MQ_INSTALLATION_PATH*/java/samples/jms

Within this directory, there are subdirectories that contain one or more sample applications as shown in Table 12 on page 115.

Table 12. IBM MQ classes for JMS sample applications

Name of sample	Description
JmsBrowser.java	A JMS queue browser application that looks at all available messages on the named queue, without removing them, in the order they would be received by a consumer application.
JmsConsumer.java	A JMS queue browser application that looks at all available messages on the named queue, without removing them, in the order they would be received by a consumer application, by looking up the connection factory instance and the destination instance in an initial context (This sample supports file system context only).

Table 12. IBM MQ classes for JMS sample applications (continued)	
Name of sample	**Description**
JmsJndiConsumer.java	A JMS consumer (receiver or subscriber) application that receives a message from the named destination (queue or topic) by looking up the connection factory instance and the destination instance in an initial context (This sample supports file system context only).
JmsJndiProducer.java	A JMS producer (sender or publisher) application that sends a simple message to the named destination (queue or topic) by looking up the connection factory instance and the destination instance in an initial context (This sample supports file system context only).
JmsProducer.java	A JMS producer (sender or publisher) application that sends a simple message to the named destination (queue or topic).
/interactive/	
SampleConsumerJava.java	Receive message(s) from a topic/queue.
SampleProducerJava.java	Send message(s) to a topic/queue.
/interactive/helper/	
BaseOptions.java	An abstract class that can be extended to provide user option(s) functionality.
IsValidType.java	Abstract class for validity checker classes.
JmsApp.java	An abstract class that can be extended to provide consumer/producer functionality.
Keys.java	A set of keys that define options for the sample applications.
Literals.java	A set of constant literals.
MyContext.java	The context in which options are presented.
Options.java	Provides functionality for user option(s).
OptionsPresenter.java	Context in which current options are presented.
/simple/	
SimpleAsyncPutPTP.java	A simple application for point-to-point messaging; message is sent asynchronously (also known as *fire-and-forget* messaging). No messages are received.
SimpleDurableSub.java	A simple application that demonstrates durable subscription facility.
SimpleJNDILookup.java	A minimal and simple application that demonstrates lookup of JMS objects using the initial context. No connection to the queue manager is made and no messages are sent or received.
SimpleMQMDRead.java	A simple application that demonstrates how a JMS application may avail MQ Message Descriptor (MQMD) fields as JMS message properties. No messages are sent; it is assumed that the queue in use is populated with some messages.

Table 12. IBM MQ classes for JMS sample applications (continued)	
Name of sample	**Description**
SimpleMQMDWrite.java	A simple application that demonstrates how a JMS application may write MQ Message Descriptor (MQMD) fields. No messages are received.
SimplePTP.java	A minimal and simple application for point-to-point messaging.
SimplePubSub.java	A minimal and simple application for publish-subscribe messaging.
SimpleReadAheadPTP.java	A simple application for point-to-point messaging; messages are streamed from the queue manager (also known as the read-ahead facility). No messages are sent; it is assumed that the queue in use is populated with some messages.
SimpleRequestor.java	A simple application that uses a requestor to send a request message and then wait for, and receive, the reply. Note: It is assumed that some other application will process the request message and send the reply message.
SimpleResponder.java	A simple application that listens on a destination for a message and then sends a reply to the message's replyTo destination. The application is written to operate in conjunction with the SimpleRequestor sample.
SimpleRetainedPub.java	A simple application that demonstrates a retained publication. No messages are received.
SimpleWMQJMSPTP.java	A minimal and simple application for point-to-point messaging.
SimpleWMQJMSPubSub.java	A minimal and simple application for publish/subscribe messaging.

The IBM MQ classes for JMS provide a script called `runjms` that can be used to run the sample applications. This script sets up the IBM MQ environment to allow you to run the IBM MQ classes for JMS sample applications.

Table 13 on page 117 shows the location of the script on each platform:

Table 13. Location of the `runjms` script	
Platform	**Directory**
AIX AIX Linux Linux	*MQ_INSTALLATION_PATH*`/java/bin/runjms`
Windows Windows	*MQ_INSTALLATION_PATH*`\java\bin\runjms.bat`

IBM i IBM i	`/qibm/proddata/mqm/java/bin/runjms`
	or
	`/qibm/proddata/mqm/java/bin/runjms64`

z/OS z/OS	*MQ_INSTALLATION_PATH*`/java/bin/runjms`

To use the `runjms` script to invoke a sample application, complete the following steps:

Procedure

1. Bring up a command prompt and navigate to the directory containing the sample application that you want to run.
2. Enter the following command:

   ```
   Path to the runjms script/runjms sample_application_name
   ```

   The sample application displays a list of parameters that it needs.
3. Enter the following command to run the sample with these parameters:

   ```
   Path to the runjms script/runjms sample_application_name parameters
   ```

Example

Linux For example, to run the `JmsBrowser` sample on Linux, enter the following commands:

```
cd /opt/mqm/samp/jms/samples
/opt/mqm/java/bin/runjms JmsBrowser -m QM1 -d LQ1
```

Related concepts

"What is installed for IBM MQ classes for JMS" on page 85
A number of files and directories are created when you install IBM MQ classes for JMS. On Windows, some configuration is performed during installation by automatically setting environment variables. On other platforms, and in certain Windows environments, you must set environment variables before you can run IBM MQ classes for JMS applications.

Scripts provided with IBM MQ classes for JMS/Jakarta Messaging

A number of scripts are provided to assist with common tasks that need to be performed when using IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

Table 14 on page 118 lists all the scripts and their uses. The scripts are in the `bin` subdirectory of the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging installation directory.

Table 14. Scripts provided with IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging

Utility	Use
Cleanup [1]	This script is maintained for compatibility with previous releases but performs no function. Manual cleanup of subscription information is no longer necessary.
DefaultConfiguration	Runs the default configuration application on platforms other than Windows.
formatLog [1]	This script is maintained for compatibility with previous releases but performs no function. Log output is now produced in readable text.
IVTRun [1] IVTSetup [1] IVTTidy [1]	Used in the point-to-point installation verification test, as described in "The point-to-point IVT for IBM MQ classes for JMS" on page 107.
JMS 3.0 JMS30Admin [1]	Runs the IBM MQ Jakarta Messaging administration tool, as described in Starting the administration tool.
JMS 3.0 JMS30Admin.config	The configuration file for the IBM MQ Jakarta Messaging administration tool, as described in Configuring the JMS administration tool.
JMS 2.0 JMSAdmin [1]	Runs the IBM MQ JMS administration tool, as described in Starting the administration tool.

Table 14. Scripts provided with IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging (continued)

Utility	Use
JMS 2.0 JMSAdmin.config	The configuration file for the IBM MQ JMS administration tool, as described in Configuring the JMS administration tool.
PSIVTRun [1]	Runs the publish/subscribe installation verification test program, as described in "The publish/subscribe IVT for IBM MQ classes for JMS" on page 111.
PSReportDump.class	This class is maintained for compatibility with previous releases, but performs no function.
JMS 3.0 setjms30env "2" on page 119	Linux AIX For Jakarta Messaging 3.0, sets the environment variables for running an IBM MQ classes for JMS application in a 32-bit Java virtual machine (JVM) on AIX and Linux systems, as described in "Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90.
JMS 2.0 setjmsenv "2" on page 119	Linux AIX For JMS 2.0, sets the environment variables for running an IBM MQ classes for JMS application in a 32-bit Java virtual machine (JVM) on AIX and Linux systems, as described in "Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90.
JMS 3.0 setjms30env64 "2" on page 119	Linux AIX For Jakarta Messaging 3.0, sets the environment variables for running an IBM MQ classes for JMS application in a 64-bit JVM on AIX and Linux systems, as described in "Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90.
JMS 2.0 setjmsenv64 "2" on page 119	Linux AIX For JMS 2.0, sets the environment variables for running an IBM MQ classes for JMS application in a 64-bit JVM on AIX and Linux systems, as described in "Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90.

Note:

1. On Windows, the file name has the extension `.bat`.
2. These scripts are available on AIX and Linux only. On Windows, after installing IBM MQ, run the command **setmqenv**. For more information, see "Setting environment variables for IBM MQ classes for JMS/Jakarta Messaging" on page 90.

Support for OSGi with IBM MQ classes for JMS

OSGi provides a framework that supports the deployment of applications as bundles. The OSGi bundles are supplied as part of the IBM MQ classes for JMS.

The IBM MQ classes for JMS includes the following OSGi bundles.

com.ibm.msg.client.osgi.jms*version_number*.jar
The common layer of code in the IBM MQ classes for JMS. For information about the layered architecture of IBM MQ classes for JMS, see IBM MQ classes for JMS architecture.

com.ibm.msg.client.osgi.jms.prereq_*version_number*.jar
The prerequisite Java archive (JAR) files for the common layer.

com.ibm.msg.client.osgi.commonservices.j2se_*version_number*.jar
Common services for Java Platform, Standard Edition (Java SE) applications.

com.ibm.msg.client.osgi.nls_*version_number*.jar
Messages for the common layer.

com.ibm.msg.client.osgi.wmq_*version_number*.jar
The IBM MQ messaging provider in IBM MQ classes for JMS. For information about the layered architecture of IBM MQ classes for JMS, see IBM MQ classes for JMS architecture.

com.ibm.msg.client.osgi.wmq.prereq_*version_number*.jar
The prerequisite JAR files for the IBM MQ messaging provider.

com.ibm.msg.client.osgi.wmq.nls_*version_number*.jar
Messages for the IBM MQ messaging provider.

com.ibm.mq.jakarta.osgi.allclient_*version_number*.jar
JMS 3.0 For Jakarta Messaging 3.0, this JAR file allows applications to use both the IBM MQ classes for JMS and the IBM MQ classes for Java, and also includes the code to handle PCF messages.

com.ibm.mq.jakarta.osgi.allclientprereqs_*version_number*.jar
JMS 3.0 For Jakarta Messaging 3.0, this JAR file provides the prerequisites for `com.ibm.mq.jakarta.osgi.allclient_version_number.jar`.

com.ibm.mq.osgi.allclient_*version_number*.jar
JMS 2.0 For JMS 2.0, this JAR file allows applications to use both the IBM MQ classes for JMS and the IBM MQ classes for Java, and also includes the code to handle PCF messages.

com.ibm.mq.osgi.allclientprereqs_*version_number*.jar
JMS 2.0 For JMS 2.0, this JAR file provides the prerequisites for `com.ibm.mq.osgi.allclient_version_number.jar`.

where *version_number* is the version number of IBM MQ that is installed.

The bundles are installed into the `java/lib/OSGi` subdirectory of your IBM MQ installation, or the `java\lib\OSGi` folder on Windows.

From IBM MQ 8.0, use the bundles `com.ibm.mq.osgi.allclient_8.0.0.0.jar`, and `com.ibm.mq.osgi.allclientprereqs_8.0.0.0.jar` for any new applications. Using these bundles removes the restriction of not being able to run both IBM MQ classes for JMS and the IBM MQ classes for Java within the same OSGi framework, all other restrictions still apply however.

The bundle `com.ibm.mq.osgi.java`*version_number*`.jar`, which is also installed into the `java/lib/OSGi` subdirectory of your IBM MQ installation, or the `java\lib\OSGi` folder on Windows, is part of the IBM MQ classes for Java. This bundle must not be loaded into an OSGi runtime environment that has the IBM MQ classes for JMS loaded.

The OSGi bundles for the IBM MQ classes for JMS have been written to the OSGi Release 4 specification. They do not work in an OSGi Release 3 environment.

You must set your system path or library path correctly so that the OSGi runtime environment can find any required DLL files or shared libraries.

If you use the OSGi bundles for the IBM MQ classes for JMS, temporary topics do not work. In addition, channel exit classes written in Java are not supported because of an inherent problem in loading classes in a multiple class loader environment such as OSGi. A user bundle can be aware of the IBM MQ classes for JMS bundles, but the IBM MQ classes for JMS bundles are not aware of any user bundle. As a result, the class loader used in an IBM MQ classes for JMS bundle cannot load a channel exit class that is in a user bundle.

For more information about OSGi, see the OSGi Alliance website.

z/OS MQ Adv. VUE JMS/Jakarta Messaging client connectivity to batch applications running on z/OS

Under certain conditions, an IBM MQ classes for JMS/Jakarta Messaging application on z/OS can connect to a queue manager on z/OS by using a client connection. Use of a client connection can simplify IBM MQ topologies.

By using a client connection, an IBM MQ classes for JMS/Jakarta Messaging application can connect to a remote z/OS queue manager if the following conditions apply:

- The application is running in a batch environment.
- The queue manager being connected to is running with IBM MQ Advanced for z/OS Value Unit Edition entitlement, and therefore has the **ADVCAP** parameter set to ENABLED.

For more information on IBM MQ Advanced for z/OS Value Unit Edition see IBM MQ product identifiers and export information.

See DISPLAY QMGR for more information on **ADVCAP** and START QMGR for more information on **QMGRPROD**.

Note that batch is the only environment supported; there is no support for JMS/Jakarta Messaging for CICS or JMS/Jakarta Messaging for IMS.

An IBM MQ classes for JMS/Jakarta Messaging application on z/OS cannot use a client mode connection to connect to a queue manager that is not running on z/OS.

If an IBM MQ classes for JMS/Jakarta Messaging application on z/OS attempts to connect using client mode, and is not allowed to do so, exception message JMSFMQ0005 is issued.

Advanced Message Security (AMS) support

IBM MQ classes for JMS/Jakarta Messaging client applications can use AMS when connecting to remote z/OS queue managers, subject to the conditions previously described in this topic.

To use AMS in this way, the client applications must use a key store type of `jceracfks` in `keystore.conf`, where:

- The property name prefix is `jceracfks` and this name prefix is case insensitive.
- The key store is a RACF keyring.
- Passwords are not required, and will be ignored. This is because RACF keyrings do not use passwords.
- If you specify the provider, the provider must be `IBMJCE`.

When you use `jceracfks` with AMS, the key store must be in the form: `safkeyring://`*`user`*`/`*`keyring`*, where:

- `safkeyring` is a literal and this name is case insensitive
- *`user`* is the RACF user id that owns the keyring
- *`keyring`* is the name of the RACF keyring and the name of the keyring is case sensitive

The following example uses the standard AMS keyring for user `JOHNDOE`:

```
jceracfks.keystore=safkeyring://JOHNDOE/drq.ams.keyring
```

Related concepts

"Java client connectivity to batch applications running on z/OS" on page 358

Under certain conditions, an IBM MQ classes for Java application on z/OS can connect to a queue manager on z/OS by using a client connection. Use of a client connection can simplify IBM MQ topologies.

Obtaining the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging separately

The libraries and utilities necessary for running applications using IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging are available in a self-extracting JAR file that you download from Fix Central. You do this if you want to get only these files, for example for deployment into a software management tool or for use with stand-alone client applications.

Before you begin

Before you start this task, make sure that you have a Java runtime environment (JRE) installed on your machine and that the JRE has been added to the system path.

The Java installer that is used in this installation process does not require running as root or any specific user. The only requirement is that the user it is run as has write access to the directory that you want the files to go in.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. It is not supported to use both the Jakarta Messaging 3.0 API and the JMS 2.0 API in the same application. For more information, see Using IBM MQ classes for JMS/Jakarta Messaging.

About this task

A self-extracting JAR file is used to minimize the size of the download, and the time that it takes to perform the extraction. The exact contents of this JAR file, and the sub-directories into which it extracts the files, depends on the version of IBM MQ.

When you run the self-extracting JAR file, it displays the IBM MQ license agreement, which must be accepted. It also lets you change the parent directory for the extraction.

For IBM MQ 9.3 and later, the self-extracting JAR file extracts the files into the following directory structure:

wmq/JavaEE
The IBM MQ resource adapter EAR and RAR files.

JMS 2.0 The following files are for use with JMS 2.0 and JMS 1.1 objects:

- `wmq.jmsra.ivt.ear`
- `wmq.jmsra.rar`

JMS 3.0 There is also an equivalent set for use with Jakarta Messaging 3.0 objects:

- `wmq.jakarta.jmsra.ivt.ear`. Contains Installation Verification Test files.
- `wmq.jakarta.jmsra.rar`. Contains resource adapter files.

wmq/JavaSE

wmq/JavaSE/bin
The **JMSAdmin** and **JMS30Admin** tools. Used to define JNDI entities that represent JMS or Jakarta Messaging objects.

JMS 2.0 The following files are for use with JMS 2.0 and JMS 1.1 objects:

- `JMSAdmin.bat`
- `JMSAdmin`
- `JMSAdmin.config`

JMS 3.0 There is also an equivalent set for use with Jakarta Messaging 3.0 objects:

- `JMS30Admin.bat`. A file that is used to start the tool on Windows.
- `JMS30Admin`. A script that is used to start the tool on Linux and UNIX platforms.
- `JMS30Admin.config`. A sample configuration file for the tool.

Note:

- Before the **JMSAdmin** tool was added to the self-extracting JAR file, the files in this directory were in the parent directory `wmq/JavaSE`.
- A client that is installed using the self-extracting JAR file can use the **JMSAdmin** or **JMS30Admin** tool to create Java messaging administered objects within a file system context (`.bindings` file). The client can also look up and use these administered objects.
- JMS 2.0 The **JMSAdmin** tool for use with JMS 2.0 and JMS 1.1 objects was added to the self-extracting JAR file at IBM MQ 9.2.0 Fix Pack 2 and IBM MQ 9.2.2.
- JMS 3.0 The **JMS30Admin** tool for use with Jakarta Messaging 3.0 objects was added to the self-extracting JAR file at IBM MQ 9.3.0.

wmq/JavaSE/lib

Advanced Message Security uses the following open source Bouncy Castle packages to support Cryptographic Message Syntax (CMS). See Support for non-IBM JREs with AMS.

V 9.4.0 From IBM MQ 9.4.0:

- `bcpkix-jdk18on.jar`
- `bcprov-jdk18on.jar`
- `bcutil-jdk18on.jar`

The following files each contain the classes for their specific JMS or Jakarta Messaging level:

- JMS 2.0 `com.ibm.mq.allclient.jar` (JMS 2.0 and JMS 1.1)
- JMS 3.0 `com.ibm.mq.jakarta.client.jar` (Jakarta Messaging 3.0)

Other prerequisite JAR files:

- `fscontext.jar`. Required if your application performs JNDI lookups using a file system context.
- JMS 3.0 `jakarta.jms-api.jar`. Contains the Jakarta Messaging 3.0 interface and Exception definitions.
- JMS 2.0 `jms.jar`. Contains the JMS 2.0 interface and Exception definitions.
- `org.json.jar`. Contains classes that allow the IBM MQ classes for JMS to interpret JSON-format CCDT files.
- `providerutil.jar`. Required if your application performs JNDI lookups using a file system context.

Note: Stabilized `com.ibm.mq.allclient.jar` and `com.ibm.mq.jakarta.client.jar` both contain a copy of the IBM MQ classes for Java. However, in IBM MQ 9.0, these classes are declared as functionally stabilized at the level shipped in IBM MQ 8.0. See Deprecations, stabilizations, and removals at IBM MQ 9.0.

wmq/OSGi

The IBM MQ OSGi client bundles:

- JMS 3.0 `com.ibm.mq.jakarta.osgi.allclient_V.R.M.F.jar`
- JMS 3.0 `com.ibm.mq.jakarta.osgi.allclientprereqs_V.R.M.F.jar`

- JMS 2.0 com.ibm.mq.osgi.allclient_*V.R.M.F*.jar
- JMS 2.0 com.ibm.mq.osgi.allclientprereqs_*V.R.M.F*.jar

where *V.R.M.F* is the Version, Release, Modification, and Fix Pack number.

Procedure

1. Download the IBM MQ Java / JMS client JAR file from Fix Central.
   a) Click this link: IBM MQ Java / JMS client.
   b) Find the client for your version of IBM MQ in the displayed list of available fixes.
      For example:

      ```
      release level: 9.3.0.0-IBM-MQ-Install-Java-All
      Long Term Support: 9.3.0.0 IBM MQ JMS and Java 'All Client'
      ```

      Then click the client file name and follow the download process.
2. Start the extraction from the directory to which you downloaded the file.
   To start the extraction, enter a command in the following format:

   ```
   java -jar V.R.M.F-IBM-MQ-Install-Java-All.jar
   ```

   where *V.R.M.F* is the product version number, for example `9.3.0.0`, and *V.R.M.F*`-IBM-MQ-Install-Java-All.jar` is the name of the file that was downloaded from Fix Central.
   For example, to extract the JMS client for the IBM MQ 9.4.0 release, you would use the following command:

   ```
   java -jar 9.4.0.0-IBM-MQ-Install-Java-All.jar
   ```

   Note: To carry out this installation, you must have a JRE installed on your machine and added to the system path.

   When you enter the command, the following information is displayed:

   ```
   Before you can use, extract, or install IBM MQ V9.4, you must accept
   the terms of 1. IBM International License Agreement for Evaluation of
   Programs 2. IBM International Program License Agreement and additional
   license information. Please read the following license agreements carefully.

   The license agreement is separately viewable using the
   --viewLicenseAgreement option.

   Press Enter to display the license terms now, or 'x' to skip.
   ```
3. Review and accept the license terms:
   a) To display the license, press Enter.
      Alternatively, press x to skip the display of the license.
      After the license is displayed, or immediately if you press x, the following message is displayed:

      ```
      Additional license information is separately viewable using the
      --viewLicenseInfo option.

      Press Enter to display additional license information now, or 'x' to skip.
      ```
   b) To display the additional license terms, press Enter.
      Alternatively, press x to skip the display of the additional license terms.
      After the additional license terms are displayed, or immediately if you press x, the following message is displayed:

      ```
      By choosing the "I Agree" option below, you agree to the terms of the
      license agreement and non-IBM terms, if applicable. If you do not
      agree, select "I do not Agree".

      Select [1] I Agree, or [2] I do not Agree:
      ```

c) To accept the license agreement and continue with selecting the installation directory, select 1.

Alternatively, select 2 to end the installation immediately.

If you select 1, a message similar to the following message is displayed:

```
Enter directory for product files or leave blank to accept the default value.
The default target directory is H:\downloads

Target directory for product files?
```

4. Specify the parent directory for the extraction.

The default location is the current directory.

- If you want to extract the product files to the default location, press Enter without specifying a value.
- If you want to extract the product files to a different location, specify the name of the directory into which you want to extract the files, then press Enter to start the extraction.

The directory name that you specify must not already exist, otherwise, when you start the extraction, an error is reported and no files are installed.

Provided that it does not already exist, the specified directory is created and the program files are extracted to this directory. During the installation, a new directory with the name `wmq` is created within the parent directory that you specified.

Three sub-directories, `JavaEE`, `JavaSE`, and `OSGi`, are created in the `wmq` directory with the following contents:

JavaEE

`wmq.jakarta.jmsra.ivt.ear`

`wmq.jakarta.jmsra.rar`

`wmq.jmsra.ivt.ear`

`wmq.jmsra.rar`

JavaSE

This directory contains the following sub-directories and files:

JavaSE/lib

`bcpkix-jdk18on.jar`

`bcprov-jdk18on.jar`

`bcutil-jdk18on.jar`

`com.ibm.mq.allclient.jar`

`com.ibm.mq.jakarta.client.jar`

`fscontext.jar`

`jms.jar`

`org.json.jar`

`providerutil.jar`

JavaSE/bin

`JMSAdmin.bat`

`JMSAdmin`

`JMSAdmin.config`

OSGi

`com.ibm.mq.jakarta.osgi.allclient_V.R.M.F.jar`

`com.ibm.mq.jakarta.osgi.allclientprereqs_V.R.M.F.jar`

JMS 2.0 `com.ibm.mq.osgi.allclient_V.R.M.F.jar`

JMS 2.0 `com.ibm.mq.osgi.allclientprereqs_V.R.M.F.jar`

When the extraction is completed, a confirmation message is displayed as shown in the following example:

```
Extracting files to H:\downloads\wmq
Successfully extracted all product files.
```

Allowlisting in IBM MQ classes for JMS/Jakarta Messaging

The Java object serialization and deserialization mechanism has been identified as a potential security risk. Allowlisting in IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging provide some protection against some serialization risks.

About this task

The Java object serialization and deserialization mechanism has been identified as a potential security risk because deserialization instantiates arbitrary Java objects, where there is the potential for maliciously sent data to cause various problems. One notable application of serialization is in Jakarta Messaging 3.0 and Java Message Service 2.0 ObjectMessages that use serialization to encapsulate and transfer arbitrary objects.

Serialization allowlisting is a potential mitigation against some of the risks that serialization poses. By explicitly specifying which classes can be encapsulated in, and extracted from, ObjectMessages, allowlisting provides some protection against some serialization risks.

Related concepts

"Running IBM MQ classes for JMS applications under the Java security manager" on page 102
IBM MQ classes for JMS can run with the Java security manager enabled. To run applications successfully with the Java security manager enabled, you must configure your Java Virtual Machine (JVM) with a suitable policy configuration file.

Allowlisting concepts

In IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging, there is support for allowlisting of classes in the implementation of the JMS ObjectMessage interface. This provides a potential mitigation against some of the security risks that potentially relate to the Java object serialization and deserialization mechanism.

Allowlisting in IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging

Important:

Wherever possible, the term *allowlist* has replaced the term *whitelist*. This includes some Java system property names mentioned in this topic. You do not have to change any existing configuration. The previous system property names also continue to work.

IBM MQ classes for JMS (JMS 2.0) and IBM MQ classes for Jakarta Messaging (Jakarta Messaging 3.0) support allowlisting of classes in the implementation of the JMS ObjectMessage interface.

- JMS 2.0 For IBM MQ classes for JMS, the relevant property names are **`com.ibm.mq.jms.allowlist.*`**.
- JMS 3.0 For IBM MQ classes for Jakarta Messaging, the relevant property names are **`com.ibm.mq.jakarta.jms.allowlist.*`**

The allowlist defines which Java classes might be serialized with ObjectMessage.setObject() and deserialized with ObjectMessage.getObject().

- JMS 2.0 Attempts to serialize or deserialize an instance of a class not included in the allowlist with ObjectMessage cause a javax.jms.MessageFormatException to be thrown, with a java.io.InvalidClassException as its cause.
- JMS 3.0 Attempts to serialize or deserialize an instance of a class not included in the allowlist with ObjectMessage cause a jakarta.jms.MessageFormatException to be thrown, with a java.io.InvalidClassException as its cause.

Producing the allowlist

Important: IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging cannot be distributed with an allowlist. The choice of classes to be transferred by using ObjectMessages is an application design choice and IBM MQ cannot preempt that.

For this reason, the allowlisting mechanism allows for two modes of operation:

DISCOVERY
In this mode, the mechanism produces a listing of fully qualified class names, reporting all classes that have been observed to be serialized or deserialized in ObjectMessages.

ENFORCEMENT
In this mode, the mechanism enforces allowlisting, rejecting attempts to serialize or deserialize classes that are not in the allowlist.

If you want to use this mechanism, you must initially run in DISCOVERY mode to gather the list of currently serialized and deserialized classes, review the list and use it as a basis for your allowlist. It might even be appropriate to use the list unchanged, but the list must be reviewed first before you decide to do this.

Controlling the allowlisting mechanism

Three system properties are available to control the allowlisting mechanism:

com.ibm.mq.jms.allowlist (JMS 2.0) and com.ibm.mq.jakarta.jms.allowlist (Jakarta Messaging 3.0)
This property can be specified in either of the following ways:

- The path name of the file that contains the allowlist, in file URI format (that is, starting with `file:`). In DISCOVERY mode, this file is written to by the allowlisting mechanism. The file must not exist. If the file does exist, the mechanism throws an exception rather than overwrite it. In ENFORCEMENT mode, this file is read by the allowlisting mechanism.
- A comma-separated of fully qualified class names that constitute the allowlist.

If this property is unset, the allowlist mechanism is inactive.

If you are using a Java security manager, you must ensure that the IBM MQ classes for JMS JAR files have read and write access to this file.

com.ibm.mq.jms.allowlist.discover (JMS 2.0) and com.ibm.mq.jakarta.jms.allowlist.discover (Jakarta Messaging 3.0)

- If this property is unset or set to false, the allowlist mechanism runs in ENFORCEMENT mode.
- If this property is set to true and the allowlist has been specified as a file URI, the allowlist mechanism runs in DISCOVERY mode.
- If this property is set to true and the allowlist has been specified as a list of class names, the allowlist mechanism throws a suitable exception.
- If this property is set to true and the allowlist has not been specified by using the com.ibm.mq.jms.allowlist or com.ibm.mq.jakarta.jms.allowlist property, the allowlist mechanism is inactive.
- If this property is set to true and the allowlist file already exists, the allowlist mechanism throws a java.io.InvalidClassException and entries are not added to the file.

com.ibm.mq.jms.allowlist.mode (JMS 2.0) and com.ibm.mq.jakarta.jms.allowlist.mode (Jakarta Messaging 3.0)

This string property can be specified in any of three ways:

- If this property is set to SERIALIZE, then ENFORCEMENT mode performs allowlist validation only on the ObjectMessage.setObject() method.
- If this property is set to DESERIALIZE, then ENFORCEMENT mode performs allowlist validation only on the ObjectMessage.getObject() method.
- If this property is unset, or set to any other value, then ENFORCEMENT mode performs allowlist validation on both the ObjectMessage.getObject() and the ObjectMessage.setObject() methods.

Format of the allowlist file

These are the main features of the format of the allowlist file:

- The allowlist file is in default platform file encoding with platform-appropriate line-endings.

  Note: If an allowlist file is being used, then that file is always written and read using the default file encoding for the JVM.

  This is fine if the allowlist file is generated in any of the following ways:

  – z/OS Generated by a stand-alone application running on z/OS and used by other stand-alone applications that are also running on z/OS.
  – Generated by an application running inside of WebSphere Application Server on any platform, and used by another instance of WebSphere Application Server.
  – Multi Generated by a stand-alone application running on IBM MQ for Multiplatforms, and used by other stand-alone applications running on IBM MQ for Multiplatforms, or by applications running inside of WebSphere Application Server on any platform.

  However, as WebSphere Application Server uses ASCII, and a standalone JVM uses EBCDIC, there will be file encoding issues if the allowlist file is generated in either of the following ways:

  – Generated on z/OS, then used by standalone applications running on a platform other than z/OS or by WebSphere Application Server.
  – Generated by either WebSphere Application Server or a standalone application running on a platform other than z/OS, then used by a stand-alone application on z/OS.
- Each non-empty line contains a fully qualified class name. Empty lines are ignored.
- Comments can be included - anything following a '#' character, to the end of the line, is ignored.
- There is a very basic wildcarding mechanism:

  – '*' can be the **last** element of a class name.
  – '*' matches a **single** element of a class name, that is, the class, but no part of the package.

  So `com.ibm.mq.*` would match `com.ibm.mq.MQMessage` but not `com.ibm.mq.jmqi.remote.api.RemoteFAP`.

  Wildcarding does not work for classes in the default package that is for classes without an explicit package name, so a class name of "*" is rejected.
- Badly formatted allowlist files, for example, files that contain an entry such as `com.ibm.mq.*.Message`, where the wildcard is not the last element, result in a java.lang.IllegalArgumentException being thrown.
- An empty allowlist file has the effect of totally disabling the use of ObjectMessage.

Format of the allowlist as a comma-separated list

The same wildcarding mechanism is available for an allowlist as a comma-separated list.

- The '*' can be expanded by the operating system if specified on a command line or in a shell script or batch file, so it might need special handling.
- The '#' comment character is only applicable when a file is specified. If the allowlist is specified as a comma-separated list of class names, then assuming that the operating system or shell doesn't process it, as it is the default comment character in many AIX and Linux shells, it is treated as a normal character.

When does allowlisting happen?

Allowlisting is initiated when the application first runs an ObjectMessage setMessage() or getMessage() method.

The system properties are evaluated, the allowlist file is opened and in ENFORCEMENT mode, the list of allowlisted classes are loaded when the mechanism is initialized. At this point, an entry is written in to the IBM MQ JMS log file for the application.

When the mechanism is initialized, its parameters might not be changed. As the time of initialization is not easily predicted as it depends on application behavior. The system property settings and the allowlist file contents should therefore be regarded as fixed from the time that the application is started. Do not change the properties or the contents of the allowlist file while the application is running, as the results are not guaranteed.

Points to consider

The best approach to mitigating the risks intrinsic to the Java serialization mechanism would be to explore alternative approaches to data transfer such as using JSON instead of ObjectMessage. Using Advanced Message Security (AMS) mechanisms can add further security by ensuring that messages come from trusted sources.

If you use the Java security manager mechanism with your application, you must grant the following permissions:

- FilePermission on any allowlist file that you use, with read permission for ENFORCEMENT mode, write permission for DISCOVER mode.
- JMS 2.0 PropertyPermission (read) on the **com.ibm.mq.jms.allowlist**, **com.ibm.mq.jms.allowlist.discover**, and **com.ibm.mq.jms.allowlist.mode** properties.
- JMS 3.0 PropertyPermission (read) on the **com.ibm.mq.jakarta.jms.allowlist**, **com.ibm.mq.jakarta.jms.allowlist.discover**, and **com.ibm.mq.jakarta.jms.allowlist.mode** properties.

More information

See “Setting up and using a JMS or Jakarta Messaging allowlist” on page 130 and “Allowlisting in WebSphere Application Server ” on page 132 for more information on allowlists.

Related concepts

“Running IBM MQ classes for JMS applications under the Java security manager” on page 102

IBM MQ classes for JMS can run with the Java security manager enabled. To run applications successfully with the Java security manager enabled, you must configure your Java Virtual Machine (JVM) with a suitable policy configuration file.

Setting up and using a JMS or Jakarta Messaging allowlist

This information tells you how an allowlist works, and how you set one up using the functionality contained in the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging to generate an allowlist file, containing a list of the types of ObjectMessages that an application can process.

Before you begin

Important:

Wherever possible, the term *allowlist* has replaced the term *whitelist*. This includes some Java system property names mentioned in this topic. You do not have to change any existing configuration. The previous system property names also continue to work.

Before starting this task, make sure that you have read and understood "Allowlisting concepts" on page 126

About this task

Because JMS and Jakarta Messaging share much in common, further references to JMS in this topic can be taken as referring to both. Any differences are highlighted as necessary.

When you have enabled the allowlisting functionality, the IBM MQ classes for JMS use that functionality in the following ways:

- When an application wants to send an ObjectMessage, it can create it in one of two ways, by calling the:
  - Session.createObjectMessage(Serializable) method, passing in the object that is to be contained within the message.
  - Session.createObjectMessage() method, to create an empty ObjectMessage, and then calling ObjectMessage.setObject(Serializable) to store the object to be sent inside the ObjectMessage.

  When either the Session.createObjectMessage(Serializable) or the ObjectMessage.setObject(Serializable) methods are called, the classes for JMS check whether the object passed in is of a type that is mentioned in the allowlist.

  If it is of a type mentioned, the object is serialized and stored within the ObjectMessage. However, if the object is of a type that is not in the allowlist, the IBM MQ classes for JMS throw a JMSException containing the message:

  ```
  JMSCC0052: An exception occurred while serializing the object:
  'java.io.InvalidClassException: <object class>; The class may not be serialized
  or deserialized as it has not been included in the allowlist '<allowlist>'.
  ```

  back to the application.

  Important: If the exception is thrown from the Session.createObjectMessage(Serializable) method, the ObjectMessage will not be created. Similarly, if the JMSException is thrown from the ObjectMessage.setObject(Serializable) method, the object will not be added to the ObjectMessage.
- If an application receives an ObjectMessage, it calls the method ObjectMessage.getObject() to get the object contained within it. When this method is called, the IBM MQ classes for JMS check the type of object contained within the ObjectMessage, to see if that object is of a type specified in the allowlist.

  If it is, the object is deserialized and returned to the application. However, if the object is of a type that is not in the allowlist, the IBM MQ classes for JMS throw a JMSException containing the message:

  ```
  JMSCC0053: An exception occurred while deserializing a message:
  'java.io.InvalidClassException: <object class>; The class may not be
  serialized or deserialized as it has not been included in the
  allowlist '<allowlist>'.'.
  ```

  back to the application.

For example, suppose your application contains the following code to send an ObjectMessage containing an object of type java.net.URI:

```
java.net.URL testURL = new java.net.URL("https://www.ibm.com/");
ObjectMessage msg = session.createObjectMessage(testURL);
sender.send(msg);
```

As allowlisting is not enabled, the application is able to successfully put the message to the required destination.

If you create a file called `C:\allowlist.txt` containing a single entry, `java.net.URL`, and you start the application again with the Java system property set:

```
-Dcom.ibm.mq.jms.allowlist=file:/C:/allowlist.txt
```

the allowlist functionality is enabled. The application is still able to create and send the ObjectMessage containing an object of type java.net.URI , as that type is specified in the allowlist.

However, if you change the `allowlist.txt` file so that the file contains the single entry `java.util.Calendar`, as the allowlist functionality is still enabled, when the application calls:

```
ObjectMessage msg = session.createObjectMessage(testURL);
```

the IBM MQ classes for JMS check the allowlist and find that it does not contain an entry for java.net.URI.

As a result, a JMSException containing the JMSCC0052 message is thrown.

Similarly, suppose you have another application that receives ObjectMessages using this code:

```
ObjectMessage message = (ObjectMessage)receiver.receive(30000);
if (message != null) {
    Object messageBody = objectMessage.getObject();
    if (messageBody instanceof java.net.URI) {
      :    :    :    :    :    :    :
```

If allowlisting is not enabled, the application is able to receive ObjectMessages that contain an object of any type. The application then checks if the object is of type java.net.URL before performing the appropriate processing.

If you now start the application with the Java system property:

```
-Dcom.ibm.mq.jms.allowlist=java.net.URL
```

set, the allowlisting functionality is turned on. When the application calls:

```
Object messageBody = objectMessage.getObject();
```

the ObjectMessage.getObject() method only returns objects of type java.net.URL.

If the object contained within the ObjectMessage is not of this type, the ObjectMessage.getObject() method throws a JMSException containing the JMSCC0053 message. The application then needs to decide what do to with the message; for example, the message could be moved to the dead-letter queue for that queue manager.

The application only returns normally if the object inside the ObjectMessage is of the type java.net.URL.

Procedure

1. Run the application which processes ObjectMessages, with the following Java system properties specified:

   ```
   -Dcom.ibm.mq.jms.allowlist.discover=true
   -Dcom.ibm.mq.jms.allowlist=file:/<path to your allowlist file>
   ```

When the application runs, the IBM MQ classes for JMS create a file which contained the types of objects that the application processed.

2. After the application has processed a representative sample of ObjectMessages over a period of time, stop it.

   The allowlist file now contains a list of all of the types of objects contained within the ObjectMessages that the application processed while it was running.

   If you have run the application for a sufficient time, this list includes all the possible types of objects contained within ObjectMessages that the application is likely to handle.

3. Restart the application with the following system property set:

   ```
   -Dcom.ibm.mq.jms.allowlist=file:/<path to your allowlist file>
   ```

   This enables allowlisting, and if the IBM MQ classes for JMS detect an ObjectMessage of a type which is not in the allowlist, a JMSException containing either the JMSCC0052 or JMSCC0053 message is thrown.

Allowlisting in WebSphere Application Server

How you use IBM MQ classes for JMS allowlisting in WebSphere Application Server.

Important:

Wherever possible, the term *allowlist* has replaced the term *whitelist*. This includes some Java system property names mentioned in this topic. You do not have to change any existing configuration. The previous system property names also continue to work.

You must ensure that your WebSphere Application Server installation includes a version of the IBM MQ resource adapter that supports allowlisting.

See "Using IBM MQ and WebSphere Application Server together" on page 481 for further information on using the two products.

IBM MQ 9.0.0 Fix Pack 1 onwards include the appropriate functionality.

Once the application server has been updated, you can use the Java system properties:

- `-Dcom.ibm.mq.jms.allowlist`
- `-Dcom.ibm.mq.jms.allowlist.discover`

described in "Setting up and using a JMS or Jakarta Messaging allowlist" on page 130.

Note: You need to set the Java system properties as generic JVM arguments, on the Java Virtual Machine used to run the application server, and the application server restarted for the changes to take effect.

See the section on *Generic JVM arguments* in Java virtual machine settings for more information.

To set the properties, go to the Java Virtual Machine window in *Process definitions* and enter the appropriate argument.

The following setting:

```
-Dcom.ibm.mq.jms.allowlist=<youruserId>_MyObject
```

causes the application server to use the allowlist *youruserId_MyObject*. Only objects of the type are processed by the application server.

The following settings:

```
-Dcom.ibm.mq.jms.allowlist.discover=true
-Dcom.ibm.mq.jms.allowlist=file:C/:allowlist.txt
```

configure the application server to use *Discover* mode, and record details of the JMS ObjectMessages, that the application server processes, to the file `C:\allowlist.txt`

The following setting:

```
-Dcom.ibm.mq.jms.allowlist=file:C/:allowlist.txt
```

causes the application server to load the file `C:/allowlist.txt`, and use the information in that file to determine the allowlist.

Related concepts

"Running IBM MQ classes for JMS applications under the Java security manager" on page 102
IBM MQ classes for JMS can run with the Java security manager enabled. To run applications successfully with the Java security manager enabled, you must configure your Java Virtual Machine (JVM) with a suitable policy configuration file.

Character string conversions in IBM MQ classes for JMS

The IBM MQ classes for JMS use CharsetEncoders and CharsetDecoders directly for character string conversion. The default behavior for character string conversion can be configured with two system properties. The handling of messages that contain unmappable characters can be configured through message properties for setting the UnmappableCharacterAction and the replacement bytes.

Before IBM MQ 8.0, string conversions in IBM MQ classes for JMS was done by calling the `java.nio.charset.Charset.decode(ByteBuffer)` and `Charset.encode(CharBuffer)` methods.

Using either of these methods results in a default replacement (REPLACE) of malformed or untranslatable data. This behavior can obscure errors in applications, and lead to unexpected characters, for example ?, in translated data.

From IBM MQ 8.0, to detect such issues earlier and more effectively, the IBM MQ classes for JMS use CharsetEncoders and CharsetDecoders directly and configure the handling of malformed and untranslatable data explicitly. The default behavior is to REPORT such issues by throwing a suitable `MQException`.

Configuring

Translating from UTF-16 (the character representation used in Java) to a native character set, such as UTF-8, is termed *encoding*, while translating in the opposite direction is termed *decoding*.

Decoding takes the default behavior for `CharsetDecoders`, reporting errors by throwing an exception.

One setting is used to specify a `java.nio.charset.CodingErrorAction` to control error handling on both encoding and decoding. One other setting is used to control the replacement byte, or bytes, when encoding. The default Java replacement String will be used in decoding operations.

UnmappableCharacterAction and replacement bytes settings in IBM MQ classes for JMS

From IBM MQ 8.0, the following two properties are available for setting the UnmappableCharacterAction and the replacement bytes. The appropriate constant definitions are in `com.ibm.msg.client.wmq.WMQConstants`.

JMS_IBM_UNMAPPABLE_ACTION
Sets or gets the `CodingErrorAction` to apply when a character cannot be mapped in an encoding or decoding operation.

You should set this as `CodingErrorAction.{REPLACE|REPORT|IGNORE}.toString()` as follows:

```
public static final String JMS_IBM_UNMAPPABLE_ACTION = "JMS_IBM_Unmappable_Action";
```

JMS_IBM_UNMAPPABLE_REPLACEMENT
Sets or gets the replacement bytes to apply when a character cannot be mapped in an encoding operation.

The default Java replacement String is used in decoding operations.

```
public static final String JMS_IBM_UNMAPPABLE_REPLACEMENT = "JMS_IBM_Unmappable_Replacement";
```

The JMS_IBM_UNMAPPABLE_ACTION and JMS_IBM_UNMAPPABLE_REPLACEMENT properties can be set on destinations or messages. A value set on a message overrides the value set on the destination to which the message is being sent.

Note that JMS_IBM_UNMAPPABLE_REPLACEMENT must be set as a single byte.

System properties for setting system defaults

From IBM MQ 8.0, the following two Java system properties are available to configure default behavior regarding character string conversion.

com.ibm.mq.cfg.jmqi.UnmappableCharacterAction
Specifies the action to be taken for untranslatable data on encoding and decoding. The value can be REPORT, REPLACE, or IGNORE.

com.ibm.mq.cfg.jmqi.UnmappableCharacterReplacement
Sets or gets the replacement bytes to apply when a character cannot be mapped in an encoding operation The default Java replacement string is used in decoding operations.

To avoid confusion between Java character and native byte representations, you should specify com.ibm.mq.cfg.jmqi.UnmappableCharacterReplacement as a decimal number representing the replacement byte in the native character set.

For example, the decimal value of ?, as a native byte, is 63 if the native character set is ASCII-based, such as ISO-8859-1, while it is 111 if the native character set is EBCDIC.

Note: Note that if an MQMD or MQMessage object has either the **unmappableAction** or **unMappableReplacement** fields set, then the values of these fields take precedence over the Java system properties. This allows the values specified by the Java system properties to be overridden for each message if required.

Related concepts

“Character string conversions in IBM MQ classes for Java” on page 339
The IBM MQ classes for Java use CharsetEncoders and CharsetDecoders directly for character string conversion. The default behavior for character string conversion can be configured with two system properties. The handling of messages that contain unmappable characters can be configured through com.ibm.mq.MQMD.

Writing IBM MQ classes for JMS/Jakarta Messaging applications

After a brief introduction to the JMS model, this section provides detailed guidance on how to write IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging applications.

About this task

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. It is not supported to use both the Jakarta Messaging 3.0 API and the JMS 2.0 API in the same application. For more information, see Using IBM MQ classes for JMS/Jakarta Messaging.

Related concepts

IBM MQ classes for Jakarta Messaging: an overview

The JMS and Jakarta Messaging model

The JMS and Jakarta Messaging model defines a set of interfaces that Java applications can use to perform messaging operations. IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are both messaging providers. They define how JMS and Jakarta Messaging objects are related to IBM

MQ concepts. The JMS and Jakarta Messaging specifications expect certain JMS and Jakarta Messaging objects to be administered objects.

JMS 2.0 IBM MQ 8.0 added support for the JMS 2.0 version of the JMS standard, which introduced a simplified API, while also retaining the classic API, from JMS 1.1.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 continues to support JMS 2.0 for existing applications. It is not supported to use both the JMS 2.0 API and the Jakarta Messaging 3.0 API in the same application.

Note: For Jakarta Messaging 3.0, control of the JMS specification moves from Oracle to the Java Community Process. However, Oracle retains control of the "javax" name, which is used in other Java technologies that have not moved to the Java Community Process. So, while Jakarta Messaging 3.0 is functionally equivalent to JMS 2.0 there are some differences in naming:

- The official name for version 3.0 is Jakarta Messaging rather than Java Message Service.
- The package and constant names are prefixed with `jakarta` rather than `javax`. For example, in JMS 2.0 the initial connection to a messaging provider is a `javax.jms.Connection` object, and in Jakarta Messaging 3.0 it is a `jakarta.jms.Connection` object.

JMS 2.0 The javax.jms packages define the JMS interfaces, and a JMS provider implements these interfaces for a specific messaging product. IBM MQ classes for JMS is a JMS provider that implements the JMS interfaces for IBM MQ.

JMS 3.0 The jakarta.jms packages define the Jakarta Messaging interfaces, and a Jakarta Messaging provider implements these interfaces for a specific messaging product. IBM MQ classes for Jakarta Messaging is a Jakarta Messaging provider that implements the Jakarta Messaging interfaces for IBM MQ.

Because JMS and Jakarta Messaging share much in common, further references to JMS in this topic can be taken as referring to both. Any differences are highlighted as necessary.

Simplified API

JMS 2.0 introduced the simplified API, while also retaining the domain specific and domain independent interfaces from JMS 1.1. The simplified API reduces the number of objects that are needed to send and receive messages and consists of the following interfaces:

ConnectionFactory
A ConnectionFactory is an administered object that is used by a JMS client to create a Connection. This interface is also used in the classic API.

JMSContext
This object combines the Connection and Session objects of the classic API. JMSContext objects can be created from other JMSContext objects, with the underlying connection being duplicated.

JMSProducer
A JMSProducer is created by a JMSContext and is used to send messages to a queue or topic. The JMSProducer object causes the creation of objects that are required to send the message.

JMSConsumer
A JMSConsumer is created by a JMSContext and is used to receive messages from a topic or a queue.

The simplified API has a number of effects:

- The JMSContext object always automatically starts the underlying connection.
- JMSProducers and JMSConsumers can now work directly with message bodies, without having to get the whole message object, by using the Message's getBody method.
- Message properties can be set on the JMSProducer object, using method chaining, before sending a 'body', a messages content. The JMSProducer will handle the creation of all objects that are needed to send the message. Using JMS 2.0, properties can be set, and a message sent as follows:

```
context.createProducer().
```

```
setProperty("foo", "bar").
setTimeToLive(10000).
setDeliveryMode(NON_PERSISTENT).
setDisableMessageTimestamp(true).
send(dataQueue, body);
```

JMS 2.0 also introduced shared subscriptions where messages can be shared between multiple consumers. All JMS 1.1 subscriptions are treated as unshared subscriptions.

Classic API

The following list summarizes the main JMS interfaces of the classic API:

Destination
A destination is where an application sends messages, or it is a source from which an application receives messages, or both.

ConnectionFactory
A ConnectionFactory object encapsulates a set of configuration properties for a connection. An application uses a connection factory to create a connection.

Connection
A Connection object encapsulates an application's active connection to a messaging server. An application uses a connection to create sessions.

Session
A session is a single threaded context for sending and receiving messages. An application uses a session to create messages, message producers, and message consumers. A session is either transacted or not transacted.

Message
A Message object encapsulates a message that an application sends or receives.

MessageProducer
An application uses a message producer to send messages to a destination.

MessageConsumer
An application uses a message consumer to receive messages sent to a destination.

Figure 9 on page 136 shows these objects and their relationships.

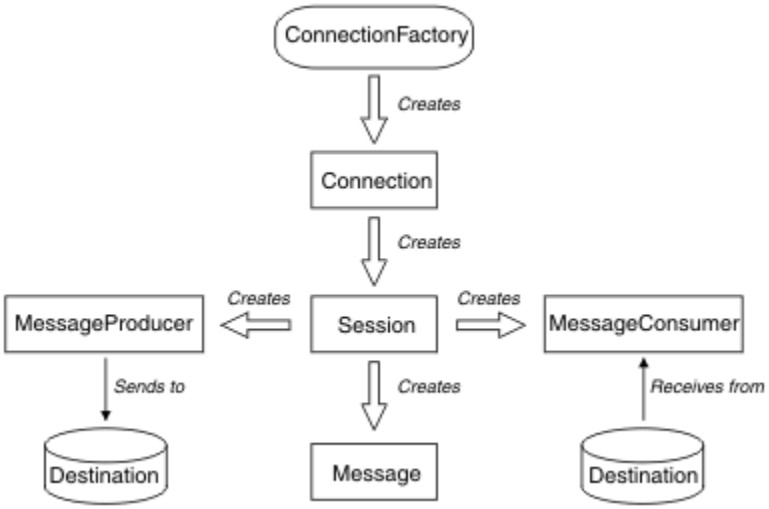


Figure 9. JMS objects and their relationships

A Destination, ConnectionFactory, or Connection object can be used concurrently by different threads of a multithreaded application, but a Session, MessageProducer, or MessageConsumer object cannot be used concurrently by different threads. The simplest way of ensuring that a Session, MessageProducer, or MessageConsumer object is not used concurrently is to create a separate Session object for each thread.

Messaging domains

JMS supports two styles of messaging:

- Point-to-point messaging
- Publish/subscribe messaging

These styles of messaging are also referred to as *messaging domains*, and you can combine both styles of messaging in an application. In the point-to-point domain, a destination is a queue and, in the publish/subscribe domain, a destination is a topic.

With versions of JMS before JMS 1.1, programming for the point-to-point domain uses one set of interfaces and methods, and programming for the publish/subscribe domain uses another set. The two sets are similar, but separate. From JMS 1.1, you can use a common set of interfaces and methods that support both messaging domains. The common interfaces provide a domain independent view of each messaging domain. Table 15 on page 137 lists the JMS domain independent interfaces and their corresponding domain specific interfaces.

Table 15. The JMS domain independent and domain specific interfaces

Domain independent interfaces	Domain specific interfaces for the point-to-point domain	Domain specific interfaces for the publish/subscribe domain
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver QueueBrowser	TopicSubscriber

JMS 2.0 IBM MQ classes for JMS 2.0 supports both the earlier JMS 1.1 domain specific interfaces and the simplified API of JMS 2.0. IBM MQ classes for JMS 2.0 can therefore be used for maintaining existing applications, including developing new function in existing applications.

JMS 3.0 IBM MQ classes for Jakarta Messaging 3.0 supports the Jakarta Messaging versions of the same interfaces, and is recommended for new application development.

In IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging, JMS objects are related to IBM MQ concepts in the following ways:

- A Connection object has properties that are derived from the properties of the connection factory that was used to create the connection. These properties control how an application connects to a queue manager. Examples of these properties are the name of the queue manager and, for an application that connects to the queue manager in client mode, the host name or IP address of the system on which the queue manager is running.
- A Session object encapsulates an IBM MQ connection handle, which therefore defines the transactional scope of the session.
- A MessageProducer object and a MessageConsumer object each encapsulates an IBM MQ object handle.

When using IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging, all the normal rules of IBM MQ apply. Note, in particular, that an application can send a message to a remote queue but it can receive a message only from a queue that is owned by the queue manager to which the application is connected.

The JMS specification expects ConnectionFactory and Destination objects to be administered objects. An administrator creates and maintains administered objects in a central repository, and a JMS application retrieves these objects using the Java Naming and Directory Interface (JNDI).

In IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging, the implementation of the Destination interface is an abstract superclass of Queue and Topic, and so an instance of Destination is either a Queue object or a Topic object. The domain independent interfaces treat a queue or a topic as a destination. The messaging domain for a MessageProducer or MessageConsumer object is determined by whether the destination is a queue or a topic.

In IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging therefore, objects of the following types can be administered objects:

- ConnectionFactory
- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic
- XAConnectionFactory
- XAQueueConnectionFactory
- XATopicConnectionFactory

Related concepts

IBM MQ Java language interfaces

"Creating and configuring connection factories and destinations" on page 197

An IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging application can create connection factories and destinations by retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace, by using the IBM JMS extensions, or by using the IBM MQ JMS extensions. An application can also use the IBM JMS extensions or IBM MQ JMS extensions to set the properties of connection factories and destinations.

JMS messages

JMS messages are composed of a header, properties, and a body. JMS defines five types of message body.

JMS messages are composed of the following parts:

Header
All messages support the same set of header fields. Header fields contain values that are used by both clients and providers to identify and route messages.

Properties
Each message contains a built-in facility to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages.

Body
JMS defines five types of message body that cover the majority of messaging styles currently in use:

Stream
A stream of Java primitive values. It is filled and read sequentially.

Map
A set of name-value pairs, where names are strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined.

Text
A message containing a java.lang.String.

Object
A message that contains a serializable Java object

Bytes
A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

The JMSCorrelationID header field is used to link one message with another. It typically links a reply message with its requesting message. JMSCorrelationID can hold a provider-specific message ID, an application-specific String, or a provider-native byte[] value.

Message selectors in JMS
Messages can contain application-defined property values. An application can use message selectors to have a JMS provider filter messages.

A message contains a built-in facility to support application-defined property values. In effect, this provides a mechanism to add application-specific header fields to a message. Properties allow an application, using message selectors, to have a JMS provider select or filter messages on its behalf, using application-specific criteria. Application-defined properties must obey the following rules:

- Property names must obey the rules for a message selector identifier.
- Property values can be Boolean, byte, short, int, long, float, double, and String.
- The JMSX and JMS_ name prefixes are reserved.

Property values are set before sending a message. When a client receives a message, the message properties are read-only. If a client attempts to set properties at this point, a MessageNotWriteableException is thrown. If clearProperties is called, the properties can now be both read from, and written to.

A property value might duplicate a value in a message body. JMS does not define a policy for what might be made into a property. However, application developers must be aware that JMS providers probably handle data in a message body more efficiently than data in message properties. For best performance, applications must use message properties only when they need to customize a message header. The primary reason for doing this is to support customized message selection.

A JMS message selector allows a client to specify the messages that it is interested in by using the message header. Only messages with headers that match the selector are delivered.

Message selectors cannot refer to message body values.

A message selector matches a message when the selector evaluates to true when the message header field and property values are substituted for their corresponding identifiers in the selector.

A message selector is a String, with syntax that 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 uppercase; however, they are not case-sensitive.

Contents of a message selector

A message selector can contain:

- Literals
  - A string literal is enclosed in quotation marks. A doubled quotation mark represents a quotation mark. Examples are 'literal' and 'literal''s'. Like Java string literals, these use the Unicode character encoding.
  - An exact numeric literal is a numeric value without a decimal point, such as 57, -957, and +62. Numbers in the range of Java long are supported.
  - 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. Numbers in the range of Java double are supported.
  - The Boolean literals TRUE and FALSE.
- Identifiers:

  - An identifier is an unlimited length sequence of Java letters and Java digits, the first of which must be a Java letter. A letter is any character for which the method Character.isJavaLetter returns true. This includes _ and $. A letter or digit is any character for which the method Character.isJavaLetterOrDigit returns true.
  - Identifiers cannot be the names NULL, TRUE, or FALSE.
  - Identifiers cannot be NOT, AND, OR, BETWEEN, LIKE, IN, or IS.
  - Identifiers are either header field references or property references.
  - Identifiers are case sensitive.
  - Message header field references are restricted to:
    - JMSDeliveryMode
    - JMSPriority
    - JMSMessageID
    - JMSTimestamp
    - JMSCorrelationID
    - JMSType

    JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType values can be null, and if so, are treated as a NULL value.
  - Any name beginning with JMSX is a JMS-defined property name.
  - Any name beginning with JMS_ is a provider-specific property name.
  - Any name that does not begin with JMS is an application-specific property name. If there is a reference to a property that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.
- White space is the same as it is defined for Java: space, horizontal tab, form feed, and line terminator.
- 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 (with a value that 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).
  - 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, with a character that 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.

Example of a message selector

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 > 2500"
```

NULL property values

As noted in the preceding list, property values can be NULL. The evaluation of selector expressions that contain NULL values is defined by SQL 92 NULL semantics. The following list gives a brief description of these semantics:

- SQL treats a NULL value as unknown.
- Comparison or arithmetic with an unknown value always yields an unknown value.
- The IS NULL operator converts an unknown value into a TRUE value.
- The IS NOT NULL operator converts an unknown value into a FALSE value.

Special behavior of JMSMessageID and JMSCorrelationID

The IBM MQ classes for JMS contain optimizations when selecting messages from a queue based on either JMSMessageID or JMSCorrelationID.

If an application specifies a selector of the form:

JMSMessageID='ID:*message_id*'

where *message_id* is a String containing a standard IBM MQ message identifier, then the IBM MQ classes for JMS use the **MatchOption** MQMO_MATCH_MSG_ID to get the message with the specified message identifier.

For example, to get a message with the message identifier 414D51207061756C745639314C545320C57C1A5F25ECE602 from a queue, an application should use the following message selector:

```
JMSMessageID='ID:414D51207061756C745639314C545320C57C1A5F25ECE602'
```

Similarly, if the application specifies a selector that has the format:

JMSCorrelationID ='ID:*correlation_id*'

where *correlation_id* is a String containing a standard IBM MQ correlation identifier, the IBM MQ classes for JMS use the **MatchOption** MQMO_MATCH_CORREL_ID to get the message with the specified correlation identifier from the queue.

In the following example, a message selector is used to get a message that has a correlation identifier of 414D51207061756C745639314C545320846E5B5F25B1CC02:

```
JMSCorrelationID='ID:414D51207061756C745639314C545320846E5B5F25B1CC02'
```

If a message selector contains a value of all zeros for either the *message_id* or *correlation_id*, then it matches any message on the queue. For example, if an application is using the selector:

```
JMSMessageID='ID:000000000000000000000000000000000000000000000000'
```

then any message on the queue is considered a match and returned to the application.

For more information about the MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID **MatchOptions**, see MatchOptions (MQLONG).

Restrictions

Although SQL supports fixed decimal comparison and arithmetic, JMS 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 alternative representation for an approximate numeric value.

SQL comments are not supported.

Mapping JMS messages onto IBM MQ messages
IBM MQ messages are composed of a Message Descriptor, an optional MQRFH2 header, and a body. The contents of a JMS message are partly mapped and partly copied to an IBM MQ message.

This topic describes how the JMS message structure that is described in the first part of this section is mapped onto an IBM MQ message. It is of interest to programmers who want to transmit messages between JMS and traditional IBM MQ applications. It is also of interest to people who want to manipulate messages transmitted between two JMS applications, for example, in an IBM Integration Bus implementation.

This section does not apply if an application uses a real-time connection to a broker. When an application uses a real-time connection, all communication is performed directly over TCP/IP; no IBM MQ queues or messages are involved.

IBM MQ messages are composed of three components:

- The IBM MQ Message Descriptor (MQMD)
- An IBM MQ MQRFH2 header
- The message body.

The MQRFH2 is optional, and its inclusion in an outgoing message is governed by the TARGCLIENT flag in the JMS Destination class. You can set this flag using the IBM MQ JMS administration tool. Because the MQRFH2 carries JMS-specific information, always include it in the message when the sender knows that the receiving destination is a JMS application. Normally, omit the MQRFH2 when sending a message directly to a non-JMS application. This is because such an application does not expect an MQRFH2 in its IBM MQ message.

If an incoming message does not have an MQRFH2 header, the Queue or Topic object derived from the JMSReplyTo header field of the message, by default, has this flag set so that a reply message sent to the queue or topic also does not have an MQRFH2 header. You can switch off this behavior of including an MQRFH2 header in a reply message only if the original message has an MQRFH2 header, by setting the TARGCLIENTMATCHING property of the connection factory to NO.

Figure 10 on page 143 shows how the structure of a JMS message is transformed to an IBM MQ message and back again:

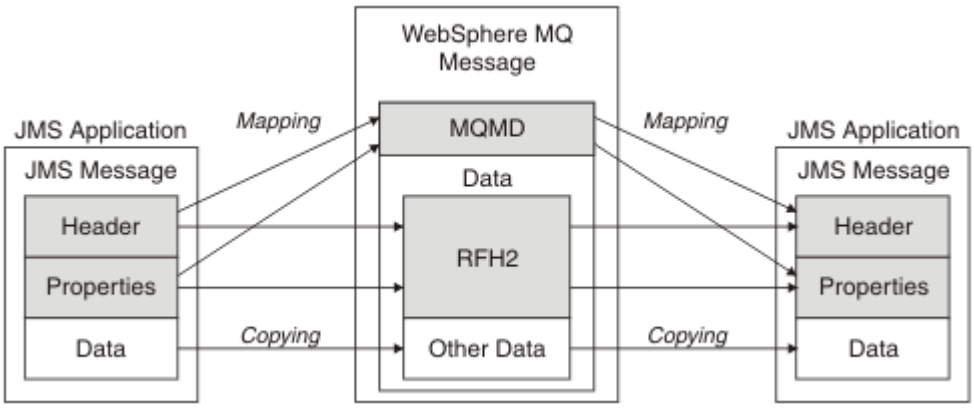


Figure 10. How messages are transformed between JMS and IBM MQ using the MQRFH2 header

The structures are transformed in two ways:

Mapping
Where the MQMD includes a field that is equivalent to the JMS field, the JMS field is mapped onto the MQMD field. Additional MQMD fields are exposed as JMS properties, because a JMS application might need to get or set these fields when communicating with a non-JMS application.

Copying
Where there is no MQMD equivalent, a JMS header field or property is passed, possibly transformed, as a field inside the MQRFH2.

The MQRFH2 header and JMS
This collection of topics describes the MQRFH Version 2 header, which carries JMS-specific data that is associated with the message content. The MQRFH Version 2 header is extensible, and can also carry additional information that is not directly associated with JMS. However, this section covers only its use by JMS. For a full description see MQRFH2 - Rules and formatting header 2.

There are two parts of the header, a fixed portion and a variable portion.

Fixed portion
The fixed portion is modeled on the *standard* IBM MQ header pattern and consists of the following fields:

StrucId (MQCHAR4)
Structure identifier.

Must be MQRFH_STRUC_ID (value: "RFH ") (initial value).

MQRFH_STRUC_ID_ARRAY (value: "R", "F", "H", " ") is also defined.

Version (MQLONG)
Structure version number.

Must be MQRFH_VERSION_2 (value: 2) (initial value).

StrucLength (MQLONG)
Total length of MQRFH2, including the NameValueData fields.

The value set into StrucLength must be a multiple of 4 (the data in the NameValueData fields can be padded with space characters to achieve this).

Encoding (MQLONG)
Data encoding.

Encoding of any numeric data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

CodedCharSetId (MQLONG)
Coded character set identifier.

Representation of any character data in the portion of the message following the MQRFH2 (the next header, or the message data following this header).

Format (MQCHAR8)
Format name.

Format name for the portion of the message following the MQRFH2.

Flags (MQLONG)
Flags.

MQRFH_NO_FLAGS =0. No flags set.

NameValueCCSID (MQLONG)
The coded character set identifier (CCSID) for the NameValueData character strings contained in this header. The NameValueData can be coded in a character set that differs from the other character strings that are contained in the header (StrucID and Format).

If the NameValueCCSID is a 2 byte Unicode CCSID (1200, 13488, or 17584), the byte order of the Unicode is the same as the byte ordering of the numeric fields in the MQRFH2. (For example, Version, StrucLength, and NameValueCCSID itself.)

Table 16. Possible values for NameValueCCSID field

CCSID	**Meaning**
1200	UTF-16, most recent Unicode version supported
13488	UTF-16, Unicode version 2.0 subset
17584	UTF-16, Unicode version 3.0 subset (includes the Euro symbol)
1208	UTF-8, most recent Unicode version supported

Variable portion
The variable portion follows the fixed portion. The variable portion contains a variable number of MQRFH2 folders. Each folder contains a variable number of elements or properties. Folders group related properties. The MQRFH2 headers created by JMS can contain any of the following folders:

The `mcd` folder

`mcd` contains properties that describe the format of the message. For example, the message service domain `Msd` property identifies a JMS message as being `JMSTextMessage`, `JMSBytesMessage`, `JMSStreamMessage`, `JMSMapMessage`, `JMSObjectMessage`, or null.

The `mcd` folder is always present in a JMS message containing an `MQRFH2`.

It is always present in a message containing an `MQRFH2` sent from IBM Integration Bus. It describes the domain, format, type, and message set of a message.

Table 17. mcd property name, synonym, data type, and folder			
Property synonym	**Property name**	**Data type**	**Folder**
	mcd.Msd	string	<mcd><Msd>*messageDomain*</Msd></mcd>
	mcd.Set	string	<mcd><Set>*messageDomain*</Set></mcd>
	mcd.Type	string	<mcd><Type>*messageDomain*</Type></mcd>
	mcd.Fmt	string	<mcd><Fmt>*messageDomain*</Fmt></mcd>

Do not add your own properties in the mcd folder.

The jms folder

jms contains JMS header fields, and JMSX properties that cannot be fully expressed in the MQMD. The jms folder is always present in a JMS MQRFH2.

The usr folder

usr contains application-defined JMS properties associated with the message. The usr folder is present only if an application has set an application-defined property.

The mqext folder

mqext contains the following types of property:

- Properties that are used only by WebSphere Application Server.
- Properties relating to delayed delivery of messages.

The folder is present if the application has either set at least one of the IBM defined properties or used delivery delay.

Table 18. mqext property name, synonym, data type, and folder			
Property synonym	**Property name**	**Data type**	**Folder**
JMSArmCorrelator	mqext.Arm	string	<mqext><Arm>*armCorrelator*</Arm></mqext>
JMSRMCorrelator	mqext.Wrm	string	<mqext><Wrm>*wrmCorrelator*</Wrm></mqext>
JMSDeliveryTime	mqext.Dlt	i8	<mqext><Dlt>*DeliveryTime*</Dlt></mqext>
JMSDeliveryDelay	mqext.Dly	i8	<mqext><Dly>*DeliveryTime*</Dly></mqext>

Do not add your own properties in the mqext folder.

The mqps folder

mqps contains properties that are used only by IBM MQ publish/subscribe. The folder is present only if the application has set at least one of the integrated publish/subscribe properties.

Table 19. mqps property name, synonym, data type, and folder			
Property synonym	**Property name**	**Data type**	**Folder**
MQTopicString	mqps.Top	string	<mqps><Top>*topicString*</Top></mqps>
MQSubUserData	mqps.Sud	string	<mqps><Sud>*subscriberUserData...*</Sud></mqps>
MQIsRetained	mqps.Ret	boolean	<mqps><Ret>*isRetained*</Ret></mqps>
MQPubOptions	mqps.Pub	i8	<mqps><Pub>*publicationOptions*</Pub></mqps>
MQPubLevel	mqps.Pbl	i8	<mqps><Pbl>*publicationLevel*</Pbl></mqps>
MQPubTime	mqpse.Pts	string	<mqps><Pts>*publicationTime*</Pts></mqps>
MQPubSeqNum	mqpse.Seq	i8	<mqps><Seq>*publicationSequenceNumber*</Seq></mqps>
MQPubStrIntData	mqpse.Sid	string	<mqps><Sid>*publicationData*</Sid></mqps>
MQPubFormat	mqpse.Pfmt	i8	<mqps><Pfmt>*messageFormat*</Pfmt></mqps>

Do not add your own properties in the mqps folder.

Table 20 on page 146 shows a full list of property names.

Table 20. MQRFH2 folders and properties used by JMS				
JMS field name	**Java type**	**MQRFH2 folder name**	**Property name**	**Type/values**
JMSDestination	Destination	jms	Dst	string
JMSExpiration	long	jms	Exp	i8
JMSPriority	int	jms	Pri	i4
JMSDeliveryMode	int	jms	Dlv	i4
JMSCorrelationID	String	jms	Cid	string
JMSReplyTo	Destination	jms	Rto	string
JMSTimestamp	long	jms	Tms	i8
JMSType	String	mcd	Type, Set, Fmt	string
JMSXGroupID	String	jms	Gid	string
JMSXGroupSeq	int	jms	Seq	i4
xxx (user defined)	Any	usr	xxx	any

Table 20. MQRFH2 folders and properties used by JMS (continued)				
JMS field name	**Java type**	**MQRFH2 folder name**	**Property name**	**Type/values**
		mcd	Msd	jms_none jms_text jms_bytes jms_map jms_stream jms_object

NameValueLength (MQLONG)
Length in bytes of the NameValueData string that immediately follows this length field (it does not include its own length).

NameValueData (MQCHARn)
A single character string, whose length in bytes is given by the preceding NameValueLength field. It contains a folder holding a sequence of properties. Each property is a name/type/value triplet, contained within an XML element whose name is the folder name, as follows:

```
<foldername>
triplet1 triplet2 .....  tripletn </foldername>
```

The closing `</foldername>` tag can be followed by spaces as padding characters. Each triplet is encoded using an XML-like syntax:

```
<name dt='datatype'>value</name>
```

The `dt='datatype'` element is optional and is omitted for many properties, because the data type is predefined. If it is included, one or more space characters must be included before the `dt=` tag.

name
is the name of the property; see Table 20 on page 146.

datatype
must match, after folding, one of the data types listed in Table 21 on page 147.

value
is a string representation of the value to be conveyed, using the definitions in Table 21 on page 147.

A null value is encoded using the following syntax:

```
<name dt='datatype' xsi:nil='true'></name>
```

Do not use `xsi:nil='false'`.

Table 21. Property data types	
Data type	**Definition**
string	Any sequence of characters excluding < and &
boolean	The character `0` or `1` (`0` = false, `1` = true)
bin.hex	Hexadecimal digits representing octets
i1	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -128 to 127 inclusive

Table 21. Property data types (continued)	
Data type	**Definition**
i2	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -32768 to 32767 inclusive
i4	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -2147483648 to 2147483647 inclusive
i8	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the range -9223372036854775808 to 9223372036854775807 inclusive
int	A number, expressed using digits `0..9`, with optional sign (no fractions or exponent). Must lie in the same range as i8. This can be used in place of one of the i* types if the sender does not want to associate a particular precision with the property
r4	Floating point number, magnitude <= 3.40282347E+38,>= 1.175E-37 expressed using digits `0..9`, optional sign, optional fractional digits, optional exponent
r8	Floating point number, magnitude <= 1.7976931348623E+308,>= 2.225E-307 expressed using digits `0..9`, optional sign, optional fractional digits, optional exponent

A string value can contain spaces. You must use the following escape sequences in a string value:

- `&` for the & character
- `<` for the < character

You can use the following escape sequences, but they are not required:

- `>` for the > character
- `&apos;` for the ' character
- `&quot;` for the " character

JMS fields and properties with corresponding MQMD fields
These tables show the MQMD fields equivalent to JMS header fields, JMS properties, and JMS provider-specific properties.

Table 22 on page 148 lists the JMS header fields and Table 23 on page 149 lists the JMS properties that are mapped directly to MQMD fields. Table 24 on page 149 lists the provider-specific properties and the MQMD fields that they are mapped to.

Table 22. JMS header fields mapping to MQMD fields			
JMS header field	**Java type**	**MQMD field**	**C type**
JMSDeliveryMode	int	Persistence	MQLONG
JMSExpiration	long	Expiry	MQLONG
JMSPriority	int	Priority	MQLONG
JMSMessageID	String	MsgID	MQBYTE24
JMSTimestamp	long	PutDate PutTime	MQCHAR8 MQCHAR8
JMSCorrelationID	String	CorrelId	MQBYTE24

Table 23. JMS properties mapping to MQMD fields			
JMS property	**Java type**	**MQMD field**	**C type**
JMSXUserID	String	UserIdentifier	MQCHAR12
JMSXAppID	String	PutApplName	MQCHAR28
JMSXDeliveryCount	int	BackoutCount	MQLONG
JMSXGroupID	String	GroupId	MQBYTE24
JMSXGroupSeq	int	MsgSeqNumber	MQLONG

Table 24. JMS provider-specific properties mapping to MQMD fields			
JMS provider-specific property	**Java type**	**MQMD field**	**C type**
JMS_IBM_Report_Exception	int	Report	MQLONG
JMS_IBM_Report_Expiration	int	Report	MQLONG
JMS_IBM_Report_COA	int	Report	MQLONG
JMS_IBM_Report_COD	int	Report	MQLONG
JMS_IBM_Report_PAN	int	Report	MQLONG
JMS_IBM_Report_NAN	int	Report	MQLONG
JMS_IBM_Report_Pass_Msg_ID	int	Report	MQLONG
JMS_IBM_Report_Pass_Correl_ID	int	Report	MQLONG
JMS_IBM_Report_Discard_Msg	int	Report	MQLONG
JMS_IBM_MsgType	int	MsgType	MQLONG
JMS_IBM_Feedback	int	Feedback	MQLONG
JMS_IBM_Format	String	Format "1" on page 149	MQCHAR8
JMS_IBM_PutApplType	int	PutApplType	MQLONG
JMS_IBM_Encoding	int	Encoding	MQLONG
JMS_IBM_Character_Set	String	CodedCharacterSetId "2" on page 150	MQLONG
JMS_IBM_PutDate	String	PutDate	MQCHAR8
JMS_IBM_PutTime	String	PutTime	MQCHAR8
JMS_IBM_Last_Msg_In_Group	boolean	MsgFlags	MQLONG

Note:

1. JMS_IBM_Format represents the format of the message body. This can be defined by the application setting the JMS_IBM_Format property of the message (note that there is an 8 character limit), or can default to the IBM MQ format of the message body appropriate to the JMS message type. JMS_IBM_Format maps to the MQMD Format field only if the message contains no RFH or RFH2 sections. In a typical message, it maps to the Format field of the RFH2 immediately preceding the message body.

2. JMS_IBM_Character_Set property value is a String value that contains the Java character set equivalent for the numeric CodedCharacterSetId value. MQMD field CodedCharacterSetId is a numeric value that contains the equivalent of the Java character set string specified by the JMS_IBM_Character_Set property.

Mapping JMS fields onto IBM MQ fields (outgoing messages)
These tables show how JMS header and property fields are mapped into MQMD and MQRFH2 fields at send() or publish() time.

Table 25 on page 150 shows how the JMS header fields are mapped into MQMD/RFH2 fields at send() or publish() time. Table 26 on page 150 shows how JMS properties are mapped into MQMD/RFH2 fields at send() or publish() time. Table 27 on page 151 shows how JMS provider-specific properties are mapped to MQMD fields at send() or publish() time,

For fields marked Set by Message Object, the value transmitted is the value held in the JMS message immediately before the send() or publish() operation. The value in the JMS message is left unchanged by the operation.

For fields marked Set by Send Method, a value is assigned when the send() or publish() is performed (any value held in the JMS message is ignored). The value in the JMS message is updated to show the value used.

Fields marked as Receive-only are not transmitted and are left unchanged in the message by send() or publish().

Table 25. Outgoing message field mapping

JMS header field name	**MQMD field used for transmission**	**Header**	**Set by**
JMSDestination		MQRFH2	Send Method
JMSDeliveryMode	Persistence	MQRFH2	Send Method
JMSExpiration	Expiry	MQRFH2	Send Method
JMSPriority	Priority	MQRFH2	Send Method
JMSMessageID	MsgID		Send Method
JMSTimestamp	PutDate/PutTime		Send Method
JMSCorrelationID	CorrelId	MQRFH2	Message Object
JMSReplyTo	ReplyToQ/ReplyToQMgr	MQRFH2	Message Object
JMSType		MQRFH2	Message Object
JMSRedelivered			Receive-only

Note:

1. MQMD field CodedCharacterSetId is a numeric value that contains the equivalent of the Java character set string specified by the JMS_IBM_Character_Set property.

Table 26. Outgoing message JMS property mapping

JMS property name	**MQMD field used for transmission**	**Header**	**Set by**
JMSXUserID	UserIdentifier		Send Method
JMSXAppID	PutApplName		Send Method
JMSXDeliveryCount			Receive-only
JMSXGroupID	GroupId	MQRFH2	Message Object

Table 26. Outgoing message JMS property mapping (continued)			
JMS property name	**MQMD field used for transmission**	**Header**	**Set by**
JMSXGroupSeq	MsgSeqNumber	MQRFH2	Message Object

Note:

These properties are defined as read-only by the JMS specification, and are set (in some cases optionally) by the JMS provider.

In IBM MQ classes for JMS two of these properties can be overridden by the application. To do this, ensure that the destination has been configured appropriately by setting the following properties:

1. Set the property `WMQConstants.WMQ_MQMD_MESSAGE_CONTEXT` to `WMQConstants.WMQ_MDCTX_SET_ALL_CONTEXT`.
2. Set the property `WMQConstants.WMQ_MQMD_WRITE_ENABLED` to `true`.

The following properties can be overridden by the application:

JMSXAppID
This property can be overridden by setting the property `WMQConstants.JMS_IBM_MQMD_PUTAPPLNAME` on the message - the value should be a Java String.

JMSXGroupID
This property can be overridden by setting the property `WMQConstants.JMS_IBM_MQMD_GROUPID` on the message - the value should be a byte array.

Table 27. Outgoing message JMS provider-specific property mapping			
JMS provider-specific property name	**MQMD field used for transmission**	**Header**	**Set by**
JMS_IBM_Report_Exception	Report		Message Object
JMS_IBM_Report_Expiration	Report		Message Object
JMS_IBM_Report_COA/COD	Report		Message Object
JMS_IBM_Report_NAN/PAN	Report		Message Object
JMS_IBM_Report_Pass_Msg_ID	Report		Message Object
JMS_IBM_Report_Pass_Correl_ID	Report		Message Object
JMS_IBM_Report_Discard_Msg	Report		Message Object
JMS_IBM_MsgType	MsgType		Message Object
JMS_IBM_Feedback	Feedback		Message Object
JMS_IBM_Format	Format		Message Object
JMS_IBM_PutApplType	PutApplType		Send Method
JMS_IBM_Encoding	Encoding		Message Object
JMS_IBM_Character_Set	CodedCharacterSetId		Message Object
JMS_IBM_PutDate	PutDate		Send Method
JMS_IBM_PutTime	PutTime		Send Method
JMS_IBM_Last_Msg_In_Group	MsgFlags		Message Object

Mapping JMS header fields at send() or publish()
These notes relate to the mapping of JMS fields at send() or publish().

JMSDestination to MQRFH2
This is stored as a string that serializes the salient characteristics of the destination object so that a receiving JMS can reconstitute an equivalent destination object. The MQRFH2 field is encoded as URI (see "Uniform resource identifiers (URIs)" on page 214 for details of the URI notation).

JMSReplyTo to MQMD.ReplyToQ, ReplyToQMgr, MQRFH2
The queue name is copied to the MQMD.ReplyToQ field, and the queue manager name is copied to the ReplyToQMgr fields. The destination extension information (other useful details that are kept in the destination object) is copied into the MQRFH2 field. The MQRFH2 field is encoded as a URI (see "Uniform resource identifiers (URIs)" on page 214 for details of the URI notation).

JMSDeliveryMode to MQMD.Persistence
The JMSDeliveryMode value is set by the send() or publish() Method or MessageProducer, unless the Destination Object overrides it. The JMSDeliveryMode value is mapped to the MQMD.Persistence field as follows:

- JMS value PERSISTENT is equivalent to MQPER_PERSISTENT
- JMS value NON_PERSISTENT is equivalent to MQPER_NOT_PERSISTENT

If the MQQueue persistence property is not set to WMQConstants.WMQ_PER_QDEF, the delivery mode value is also encoded in the MQRFH2.

JMSExpiration to/from MQMD.Expiry, MQRFH2
JMSExpiration stores the time to expire (the sum of the current time and the time to live), whereas MQMD stores the time to live. Also, JMSExpiration is in milliseconds, but MQMD.Expiry is in tenths of a second.

- If the send() method sets an unlimited time to live, MQMD.Expiry is set to MQEI_UNLIMITED, and no JMSExpiration is encoded in the MQRFH2.
- If the send() method sets a time to live that is less than 214748364.7 seconds (about 7 years), the time to live is stored in MQMD.Expiry, and the expiration time (in milliseconds), is encoded as an i8 value in the MQRFH2.
- If the send() method sets a time to live greater than 214748364.7 seconds, MQMD.Expiry is set to MQEI_UNLIMITED. The true expiration time in milliseconds is encoded as an i8 value in the MQRFH2.

JMSPriority to MQMD.Priority
Directly map JMSPriority value (0-9) onto MQMD priority value (0-9). If JMSPriority is set to a non-default value, the priority level is also encoded in the MQRFH2.

JMSMessageID from MQMD.MessageID
All messages sent from JMS have unique message identifiers assigned by IBM MQ. The value assigned is returned in the MQMD.MessageId field after the MQPUT call, and is passed back to the application in the JMSMessageID field. The IBM MQ messageId is a 24-byte binary value, whereas the JMSMessageID is a string. The JMSMessageID is composed of the binary messageId value converted to a sequence of 48 hexadecimal characters, prefixed with the characters ID:. JMS provides a hint that can be set to disable the production of message identifiers. This hint is ignored, and a unique identifier is assigned in all cases. Any value that is set into the JMSMessageID field before a send() is overwritten.

If you do require the ability to specify the MQMD.MessageID, you can do this with one of the IBM MQ JMS extensions described in "Reading and writing the message descriptor from an IBM MQ classes for JMS application" on page 236.

JMSTimestamp to MQRFH2
During a send, the JMSTimestamp field is set according to the JVM's clock. This value is set into the MQRFH2. Any value that is set into the JMSTimestamp field before a send() is overwritten. See also the JMS_IBM_PutDate and JMS_IBM_PutTime properties.

JMSType to MQRFH2
This string is set into the MQRFH2 mcd.Type field. If it is in URI format, it can also affect mcd.Set and mcd.Fmt fields.

JMSCorrelationID to MQMD.CorrelId, MQRFH2
The JMSCorrelationID can hold one of the following:

A provider specific message ID
This is a message identifier from a message previously sent or received, and so should be a string of 48 lowercase hexadecimal digits that are prefixed with `ID:` The prefix is removed, the remaining characters are converted into binary, and then they are set into the MQMD.CorrelId field.

A provider-native byte[] value
The value is copied into the MQMD.CorrelId field - padded with nulls, or truncated to 24 bytes if necessary. No CorrelId value is encoded in the MQRFH2.

An application-specific string
The value is copied into the MQRFH2. The first 24 bytes of the string, in UTF8 format, are written into the MQMD.CorrelID.

Mapping JMS property fields
These notes refer to the mapping of JMS property fields in IBM MQ messages.

JMSXUserID from MQMD UserIdentifier
JMSXUserID is set on return from send call.

JMSXAppID from MQMD PutApplName
JSMXAppID is set on return from send call.

JMSXGroupID to MQRFH2 (point-to-point)
For point-to-point messages, the JMSXGroupID is copied into the MQMD GroupID field. If the JMSXGroupID starts with the prefix ID:, it is converted into binary. Otherwise, it is encoded as a UTF8 string. The value is padded or truncated if necessary to a length of 24 bytes. The MQMF_MSG_IN_GROUP flag is set.

JMSXGroupID to MQRFH2 (publish/subscribe)
For publish/subscribe messages, the JMSXGroupID is copied into the MQRFH2 as a string.

JMSXGroupSeq MQMD MsgSeqNumber (point-to-point)
For point-to-point messages, the JMSXGroupSeq is copied into the MQMD MsgSeqNumber field. The MQMF_MSG_IN_GROUP flag is set.

JMSXGroupSeq MQMD MsgSeqNumber (publish/subscribe)
For publish/subscribe messages, the JMSXGroupSeq is copied into the MQRFH2 as an i4.

Mapping JMS provider-specific fields
The following notes refer to the mapping of JMS provider-specific fields into IBM MQ messages.

JMS_IBM_Report_*XXX* to MQMD Report
A JMS application can set the MQMD Report options, using the following JMS_IBM_Report_*XXX* properties. The single MQMD is mapped to several JMS_IBM_Report_*XXX* properties.

The JMS_IBM_Report_*XXX* constants are in `com.ibm.msg.client.jakarta.wmq.WMQConstants` or `com.ibm.msg.client.wmq.WMQConstants`.

JMS_IBM_Report_Exception

MQRO_EXCEPTION or
MQRO_EXCEPTION_WITH_DATA or
MQRO_EXCEPTION_WITH_FULL_DATA

JMS_IBM_Report_Expiration

MQRO_EXPIRATION or
MQRO_EXPIRATION_WITH_DATA or
MQRO_EXPIRATION_WITH_FULL_DATA

JMS_IBM_Report_COA

MQRO_COA or
MQRO_COA_WITH_DATA or
MQRO_COA_WITH_FULL_DATA

JMS_IBM_Report_COD

MQRO_COD or
MQRO_COD_WITH_DATA or
MQRO_COD_WITH_FULL_DATA

JMS_IBM_Report_PAN
MQRO_PAN

JMS_IBM_Report_NAN
MQRO_NAN

JMS_IBM_Report_Pass_Msg_ID
MQRO_PASS_MSG_ID

JMS_IBM_Report_Pass_Correl_ID
MQRO_PASS_CORREL_ID

JMS_IBM_Report_Discard_Msg
MQRO_DISCARD_MSG

The MQRO values are in `com.ibm.mq.constants.CMQC`.

JMS_IBM_MsgType to MQMD MsgType
Value maps directly onto MQMD MsgType. If the application has not set an explicit value of JMS_IBM_MsgType, a default value is used. This default value is determined as follows:

- If JMSReplyTo is set to an IBM MQ queue destination, MSGType is set to the value MQMT_REQUEST
- If JMSReplyTo is not set, or is set to anything other than an IBM MQ queue destination, MsgType is set to the value MQMT_DATAGRAM

JMS_IBM_Feedback to MQMD Feedback
Value maps directly onto MQMD Feedback.

JMS_IBM_Format to MQMD Format
Value maps directly onto MQMD Format.

JMS_IBM_Encoding to MQMD Encoding
If set, this property overrides the numeric encoding of the Destination Queue or Topic.

JMS_IBM_Character_Set to MQMD CodedCharacterSetId
If set, this property overrides the coded character set property of the Destination Queue or Topic.

JMS_IBM_PutDate from MQMD PutDate
The value of this property is set, during send, directly from the PutDate field in the MQMD. Any value that is set into the JMS_IBM_PutDate property before a send is overwritten. This field is a String of eight characters, in the IBM MQ Date format of YYYYMMDD. This property can be used with the JMS_IBM_PutTime property to determine the time the message was put according to the queue manager.

JMS_IBM_PutTime from MQMD PutTime
The value of this property is set, during send, directly from the PutTime field in the MQMD. Any value that is set into the JMS_IBM_PutTime property before a send is overwritten. This field is a String of eight characters, in the IBM MQ Time format of HHMMSSTH. This property can be used with the JMS_IBM_PutDate property to determine the time the message was put according to the queue manager.

JMS_IBM_Last_Msg_In_Group to MQMD MsgFlags
For point-to-point messaging, this Boolean value maps to the MQMF_LAST_MSG_IN_GROUP flag in the MQMD MsgFlags field. It is normally used with the JMSXGroupID and JMSXGroupSeq properties

to indicate to a legacy IBM MQ application that this message is the last in a group. This property is ignored for publish/subscribe messaging.

Mapping IBM MQ fields onto JMS fields (incoming messages)
These tables show how JMS header and property fields are mapped into MQMD and MQRFH2 fields at get() or receive() time.

Table 28 on page 155 shows how JMS header fields are mapped onto MQMD/MQRFH2 fields at get() or receive() time. Table 29 on page 155 shows how JMS property fields are mapped onto MQMD/MQRFH2 fields at get() or receive() time. Table 30 on page 156 shows how JMS provider-specific properties are mapped.

Table 28. Incoming message JMS header field mapping

JMS header field name	MQMD field retrieved from	MQRFH2 field retrieved from
JMSDestination		jms.Dst or mqps.Top "1" on page 155
JMSDeliveryMode	Persistence "2" on page 155	jms.Dlv "2" on page 155
JMSExpiration		jms.Exp
JMSPriority	Priority	
JMSMessageID	MsgID	
JMSTimestamp	PutDate "2" on page 155 PutTime "2" on page 155	jms.Tms "2" on page 155
JMSCorrelationID	CorrelId "2" on page 155	jms.Cid "2" on page 155
JMSReplyTo	ReplyToQ "2" on page 155 ReplyToQMgr "2" on page 155	jms.Rto "2" on page 155
JMSType		mcd.Type, mcd.Set, mcd.Fmt
JMSRedelivered	BackoutCount	

Note:

1. If both jms.Dst and mqps.Top are set, the value in jms.Dst is used.
2. For properties that can have values retrieved from the MQRFH2 or the MQMD, if both are available, the setting in the MQRFH2 is used.
3. JMS_IBM_Character_Set property value is a String value that contains the Java character set equivalent for the numeric CodedCharacterSetId value.

Table 29. Incoming message property mapping

JMS property name	MQMD field retrieved from	MQRFH2 field retrieved from
JMSXUserID	UserIdentifier	
JMSXAppID	PutApplName	
JMSXDeliveryCount	BackoutCount	
JMSXGroupID	GroupId "1" on page 156	jms.Gid "1" on page 156
JMSXGroupSeq	MsgSeqNumber "1" on page 156	jms.Seq "1" on page 156

Note:

1. For properties that can have values retrieved from the MQRFH2 or the MQMD, if both are available, the setting in the MQRFH2 is used. The properties are set from the MQMD values only if the MQMF_MSG_IN_GROUP or MQMF_LAST_MSG_IN_GROUP message flags are set.

Table 30. Incoming message provider-specific JMS property mapping

JMS property name	MQMD field retrieved from	MQRFH2 field retrieved from
JMS_IBM_Report_Exception	Report	
JMS_IBM_Report_Expiration	Report	
JMS_IBM_Report_COA	Report	
JMS_IBM_Report_COD	Report	
JMS_IBM_Report_PAN	Report	
JMS_IBM_Report_NAN	Report	
JMS_IBM_Report_ Pass_Msg_ID	Report	
JMS_IBM_Report_Pass_Correl_ID	Report	
JMS_IBM_Report_Discard_Msg	Report	
JMS_IBM_MsgType	MsgType	
JMS_IBM_Feedback	Feedback	
JMS_IBM_Format	Format	
JMS_IBM_PutApplType	PutApplType	
JMS_IBM_Encoding "1" on page 156	Encoding	
JMS_IBM_Character_Set "1" on page 156	CodedCharacterSetId	
JMS_IBM_PutDate	PutDate	
JMS_IBM_PutTime	PutTime	
JMS_IBM_Last_Msg_In_Group	MsgFlags	

1. Only set if the incoming message is a Bytes Message.

Exchanging messages between a JMS application and a traditional IBM MQ application
This topic describes what happens when a JMS application exchanges messages with a traditional IBM MQ application that cannot process the MQRFH2 header.

Figure 11 on page 157 shows the mapping.

The administrator indicates that the JMS application is communicating with a traditional IBM MQ application by setting the TARGCLIENT property of the destination to *MQ*. This indicates that no MQRFH2 header is to be produced. If this is not done, the receiving application must be able to handle the MQRFH2 header.

The mapping from JMS to MQMD targeted at a traditional IBM MQ application is the same as mapping from JMS to MQMD targeted at a JMS application. If IBM MQ classes for JMS receives an IBM MQ message with the MQMD *Format* field set to anything other than MQFMT_RFH2, data is being received from a non-JMS application. If the format is MQFMT_STRING, the message is received as a JMS text message. Otherwise, it is received as a JMS bytes message. Because there is no MQRFH2, only those JMS properties that are transmitted in the MQMD can be restored.

If IBM MQ classes for JMS receives a message that does not have an MQRFH2 header, the TARGCLIENT property of the Queue or Topic object derived from the JMSReplyTo header field of the message is set to MQ by default. This means that a reply message sent to the queue or topic also does not have an MQRFH2 header. You can switch off this behavior of including an MQRFH2 header in a reply message only if the original message has an MQRFH2 header, by setting the TARGCLIENTMATCHING property of the connection factory to NO.

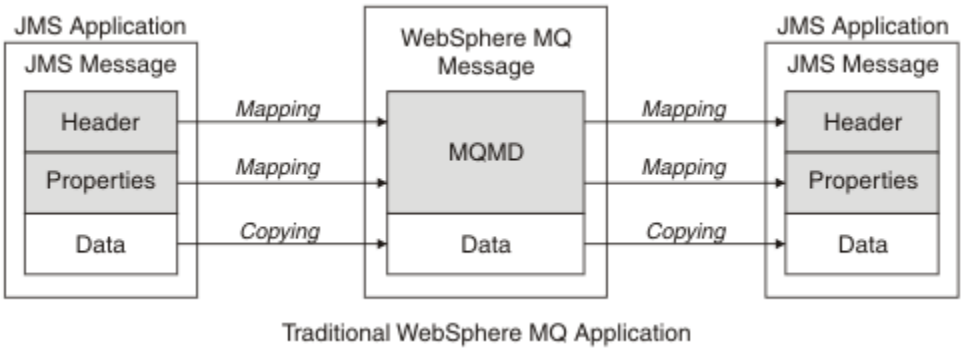


Figure 11. How JMS messages are transformed to IBM MQ messages with no MQRFH2 header

The JMS message body
This topic contains information about the encoding of the message body itself. The encoding depends on the type of JMS message.

ObjectMessage
An ObjectMessage is an object serialized by the Java Runtime in the normal way.

TextMessage
A TextMessage is an encoded string. For an outgoing message, the string is encoded in the character set given by the destination object. This defaults to UTF8 encoding (the UTF8 encoding starts with the first character of the message; there is no length field at the start). It is, however, possible to specify any other character set supported by IBM MQ classes for JMS. Such character sets are used mainly when you send a message to a non-JMS application.

If the character set is a double-byte set (including UTF16), the destination object's integer encoding specification determines the order of the bytes.

An incoming message is interpreted using the character set and encoding that are specified in the message itself. These specifications are in the last IBM MQ header (or MQMD if there are no headers). For JMS messages, the last header is usually the MQRFH2.

BytesMessage
A BytesMessage is, by default, a sequence of bytes as defined by the JMS 1.0.2 specification and associated Java documentation.

For an outgoing message that was assembled by the application itself, the destination object's encoding property can be used to override the encodings of integer and floating point fields contained in the message. For example, you can request that floating point values are stored in S/390 rather than IEEE format).

An incoming message is interpreted using the numeric encoding specified in the message itself. This specification is in the last IBM MQ header (or MQMD if there are no headers). For JMS messages, the last header is usually the MQRFH2.

If a BytesMessage is received, and is re-sent without modification, its body is transmitted byte for byte, as it was received. The destination object's encoding property has no effect on the body. The only string-like entity that can be sent explicitly in a BytesMessage is a UTF8 string. This is encoded in Java UTF8 format, and starts with a 2-byte length field. The destination object's character set property has no effect on the encoding of an outgoing BytesMessage. The character set value in an incoming IBM MQ message has no effect on the interpretation of that message as a JMS BytesMessage.

Non-Java applications are unlikely to recognize the Java UTF8 encoding. Therefore, for a JMS application to send a BytesMessage that contains text data, the application itself must convert its strings to byte arrays, and write these byte arrays into the BytesMessage.

MapMessage
A MapMessage is a string containing XML name/type/value triplets encoded as:

```
<map>
  <elt name="elementname1" dt="datatype1">value1</elt>
  <elt name="elementname2" dt="datatype2">value2</elt>
  ...
</map>
```

where `datatype` is one of the data types listed in Table 21 on page 147. The default data type is `string`, and so the attribute `dt="string"` is omitted for string elements.

The character set used to encode or interpret the XML string that forms the body of a map message is determined according to the rules that apply to a text message.

Versions of IBM MQ classes for JMS earlier than 5.3 encoded the body of a map message in the following format:

```
<map>
  <elementname1 dt="datatype1">value1</elementname1>
  <elementname2 dt="datatype2">value2</elementname2>
  ...
</map>
```

IBM MQ classes for JMS 5.3 and later can interpret either format, but versions of IBM MQ classes for JMS earlier than 5.3 cannot interpret the current format.

If an application needs to send map messages to another application that is using a version of IBM MQ classes for JMS earlier than 5.3, the sending application must call the connection factory method `setMapNameStyle(WMQConstants.WMQ_MAP_NAME_STYLE_COMPATIBLE)` to specify that the map messages are sent in the previous format. By default, all map messages are sent in the current format.

StreamMessage
A StreamMessage is like a map message, but without element names:

```
<stream>
  <elt dt="datatype1">value1</elt>
  <elt dt="datatype2">value2</elt>
  ...
</stream>
```

where `datatype` is one of the data types listed in Table 21 on page 147. The default data type is `string`, and so the attribute `dt="string"` is omitted for string elements.

The character set used to encode or interpret the XML string that makes up the StreamMessage body is determined following the rules that apply to a TextMessage.

The MQRFH2.format field is set as follows:

MQFMT_NONE
for ObjectMessage, BytesMessage, or messages with no body.

MQFMT_STRING
for TextMessage, StreamMessage, or MapMessage.

JMS message conversion
Message data conversion in JMS is performed when sending and receiving messages. IBM MQ performs most data conversion automatically. It converts text and numeric data when transferring a message

between JMS applications. Text is converted when exchanging a JMSTextMessage between a JMS application and an IBM MQ application.

If you are planning to do more complex message exchanges, the following topics are of interest you. Complex message exchanges include:

- Transferring non-text messages between an IBM MQ application and a JMS application.
- Exchanging text data in byte format.
- Converting the text in your application.

JMS message data

Data conversion is necessary to exchange text and numeric data between applications, even between two JMS applications. The internal representation of text and numbers must be encoded so they can be transferred in a message. Encoding forces a decision about how numbers and text are represented. IBM MQ manages the encoding of text and numbers in JMS messages, except for JMSObjectMessage, see "JMSObjectMessage" on page 165. It uses three message attributes. The three attributes are CodedCharacterSetId, Encoding, and Format.

These three message attributes are normally stored in the JMS header, MQRFH2, fields of a JMS message. If the message type is an MQ, rather than JMS type of message, the attributes are stored in the message descriptor, MQMD. The attributes are used to convert the JMS message data. JMS message data is transferred in the message data part of an IBM MQ message.

JMS message properties

JMS message properties, such as JMS_IBM_CHARACTER_SET, are exchanged in the MQRFH2 header part of a JMS message, unless the message has been sent without an MQRFH2. Only JMSTextMessage and JMSBytesMessage can be sent without an MQRFH2. If a JMS property is stored as an IBM MQ message property in the message descriptor, MQMD, it is converted as part of the MQMD conversion. If a JMS property is stored in the MQRFH2, it is stored in the character set specified by MQRFH2.NameValueCCSID. When a message is sent or received, message properties are converted to and from their internal representation in the JVM. The conversion is to and from the character set of the message descriptor or MQRFH2.NameValueCCSID. Numeric data is converted to text.

JMS message conversion

The following topics contain examples and tasks that are useful if you plan to exchange more complex messages that require conversion.

JMS message conversion approaches
A number of data conversion approaches are open to JMS application designers. These approaches are not exclusive; some applications are likely to use a combination of these approaches. If your application is exchanging only text or is exchanging messages only with other JMS applications, you do not normally consider data conversion. Data conversion is performed automatically for you, by IBM MQ.

You can ask a number of questions about how to approach message conversion:

Is it necessary to think about message conversion at all?
In some cases, such as JMS to JMS message transfers, and exchanging text messages with IBM MQ programs, IBM MQ performs the necessary conversions for you, automatically. You might want to control data conversion for performance reasons, or you might be exchanging complex messages that have a predefined format. In cases such as these you must understand message conversion, and read the following topics.

What kinds of conversion are there?
There are four main types of conversion, which are explained in the following sections:

1. "JMS client data conversion" on page 160
2. "Application data conversion" on page 160

3. “Queue manager data conversion” on page 161
4. “Message channel data conversion” on page 162

Where should conversion be performed?

The section, “Choosing an approach to message conversion: receiver makes good” on page 162, describes the usual approach of "receiver makes good". "Receiver makes good" also applies to JMS data conversion.

JMS client data conversion

JMS client [1] data conversion is the conversion of Java primitives and objects into bytes in a JMS message as it is sent to a destination, and conversion back again, when it is received. JMS client data conversion uses the methods of the `JMSMessage` classes. The methods are listed by `JMSMessage` class type in Table 31 on page 163.

Conversion to and from the internal JVM representation of numbers and text is performed for read, get, set, and write methods. The conversion is performed when the message is sent, and when any of the read or get methods is called on a message that has been received.

The code page and numeric encoding used to write or set the contents of a message are defined as attributes of the destination. The destination code page and numeric encoding can be changed administratively. An application can also override the destination code page and encoding by setting the message properties that control writing or setting message content.

If you want to convert number encoding when a `JMSBytesMessage` message is sent to a destination that is not defined as `Native` encoding, you must set the message property `JMS_IBM_ENCODING` before sending the message. If you are following the "receiver makes good" pattern, or if you are exchanging messages between JMS applications, the application does not need to set `JMS_IBM_ENCODING`. In most cases you can leave the `Encoding` property as `Native`.

For `JMSStreamMessage`, `JMSMapMessage`, and `JMSTextMessage` messages, the character set identifier properties of the destination are used. Encoding is ignored on sending as numbers are written out in text format. The JMS client application program does not have to set `JMS_IBM_CHARACTER_SET` before sending the message if the destination character set property to apply.

To get the data in a message an application calls the JMS message read or get methods. The methods refer to the code page and encoding defined in the previous message header to create the Java primitives and objects correctly.

JMS client data conversion meets the needs of most JMS applications that are exchanging messages between one JMS client and another. You do not code any explicit data conversion. You do not use the `java.nio.charset.Charset` class, which is typically used when writing text to a file. The `writeString` and `setString` methods do the conversion for you.

For more details on JMS client data conversion, see “JMS client message conversion and encoding” on page 172.

Application data conversion

A JMS client application can perform explicit character data conversion by using the `java.nio.charset.Charset` class; see the examples in Figure 14 on page 164 and Figure 15 on page 165. String data is converted into bytes, using the `getBytes` method, and sent as bytes. The bytes are converted back into text by using a `String` constructor that takes a byte array and a `Charset`. Character data is converted using the `encode` and `decode` `Charset` methods. Typically the message is sent or received as `JMSBytesMessage`, because the message part of a `JMSBytesMessage` does not contain anything other than the data written by the application [2] . You can also send and receive bytes using `JMSStreamMessage`, `JMSMapMessage`, or `JMSObjectMessage`.

---

[1] "JMS Client" refers to the IBM MQ classes for JMS that implement the JMS interface, which runs either in client or bindings mode.

[2] One exception: Data written using `writeUTF` starts with a 2 byte length field

There are no Java methods to encode and decode bytes that contain numeric data represented in different encoding formats. Numeric data is encoded and decoded automatically using the numeric JMSMessage read and write methods. The read and write methods use the value of the JMS_IBM_ENCODING attribute of the message data.

A typical use for application data conversion is if a JMS client sends or receives a formatted message from a non-JMS application. A formatted message contains text, numeric, and bytes data organized by the length of the data fields. Unless the non-JMS application has specified the message format as "MQSTR ", the message is constructed as a JMSBytesMessage. To receive formatted message data in a JMSBytesMessage you must call a sequence of methods. The methods must be called in the same order the fields were written into the message. If the fields are numeric, you must know the encoding and length of the numeric data. If any of the fields contain byte or text data, you must know the length of any byte data in the message. There are two ways to convert a formatted message into a Java object that is easy to use.

1. Construct a Java class corresponding to the record, to encapsulate reading and writing the message. Access to the data in the record is with get and set methods of the class.
2. Construct a Java class corresponding to the record by extending the `com.ibm.mq.headers` class. Access to the data in the class is with type-specific accessors of the form, `getStringValue(`*`fieldname`*`);`

See “Exchanging a formatted record with a non-JMS application” on page 179.

Queue manager data conversion

Code page conversion can be performed by the queue manager when a JMS client program gets a message. The conversion is the same as the conversion performed for a C program. A C program sets MQGMO_CONVERT as an MQGET GetMsgOpts parameter option; see Figure 13 on page 164. A queue manager performs conversion for a JMS client program that is receiving a message, if the WMQ_RECEIVE_CONVERSION destination property is set to WMQ_RECEIVE_CONVERSION_QMGR, The JMS client program can also set the destination property; see Figure 12 on page 161.

```
((MQDestination)destination).setIntProperty(
              WMQConstants.WMQ_RECEIVE_CONVERSION,
              WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
```

Or,

```
((MQDestination)destination).setReceiveConversion
              (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
```

Figure 12. Enable queue manager data conversion

The main benefit of queue manager conversion comes when exchanging messages with non-JMS applications. If the Format field in the message is defined, and the target character set, or encoding, is different to the message, the queue manager performs data conversion for the target application, if the application requests it. The queue manager converts message data formatted according to one of the predefined IBM MQ message types, such as a CICS bridge header (MQCIH). If the Format field is user-defined, the queue manager looks for a data conversion exit with the name provided in the Format field.

Queue manager data conversion is used to best effect with the "receiver makes good" design pattern. A sending JMS client does not need to perform conversion. A non-JMS receiving program relies on the conversion exit to ensure that the message is delivered in the required code page and encoding. With a sending JMS client, and non-JMS receiver, the example applies to IBM MQ.

You can create a data conversion exit, using the data conversion exit utility, **crtmqcvx**, to enable the queue manager to convert your own record formatted data. You can build your own record format, use the

com.ibm.mq.headers to access it as a Java class, and use your own conversion exit to convert it. On z/OS the utility is called **CSQUCVX**, and on IBM i, **CVTMQMDTA**. See “Exchanging a formatted record with a non-JMS application” on page 179.

Message channel data conversion

IBM MQ Sender, Server, Cluster-receiver, and Cluster-sender channels have a message conversion option, CONVERT. The contents of a message can optionally be converted when a message is sent. The conversion takes place at the sending end of the channel. The cluster-receiver definition is used to auto-define the corresponding cluster-sender channel.

Data conversion by message channels is typically used if it is not possible to use other forms of conversion.

Choosing an approach to message conversion: "receiver makes good"

The usual approach in IBM MQ application design for code conversion is "receiver makes good". "Receiver makes good" reduces the number of message conversions. It also avoids the problem of unexpected channel errors if message conversion fails on some intermediary queue manager during message transfer. The "receiver makes good" rule is only broken if there is some reason why the receiver cannot make good. The receiving platform might not have the right character set, for example.

"Receiver makes good" is also good general guidance for JMS client applications. But in specific cases, conversion to the correct character set at source can be more efficient. Conversion from the JVM internal representation must take place when a message containing text or numeric types is sent. Conversion to the character set required by the receiver, if the receiver is not a JMS client, might remove the need for the non-JMS recipient to perform conversion. If the recipient is a JMS client, it is going to convert again, anyway, to decode the message data and create Java primitives and objects.

The difference between JMS client applications, and applications written in a language such as C, is that Java must perform data conversion. A Java application must convert numbers and text from their internal representation to an encoded format used in messages.

By setting destination, or message properties, you can set the character set and encoding used by IBM MQ to encode numbers and text in messages. Normally, you would leave the character set as 1208 and encoding as Native.

IBM MQ does not convert byte arrays. To encode strings and character arrays into byte arrays use the java.nio.charset package. Charset specifies the character set used to convert a string or character array into a byte array. You can also decode a byte array into a string or character array using a Charset. It is not good practice to rely on java.nio.charset.Charset.defaultCodePage when encoding strings and character arrays. The default Charset is typically windows-1252 on Windows, and UTF-8 on AIX and Linux. windows-1252 is a single-byte character set and UTF-8 is a multi-byte character set.

Generally leave the destination character set and encoding properties at their default values of UTF-8 and Native when exchanging messages with other JMS applications. If you are exchanging messages containing numbers or text with a JMS application, choose one of the JMSTextMessage, JMSStreamMessage, JMSMapMessage, or JMSObjectMessage message types that fit your purpose. There are no other conversion tasks to do.

If you are exchanging messages with non-JMS applications that use a record format, it is more complicated. Unless the entire record contains text and can be transferred as a JMSTextMessage, you must encode and decode text in the application. Set the destination message type to MQ, and use JMSBytesMessage to avoid the IBM MQ classes for JMS adding additional header and tagging information to the message data. Use JMSBytesMessage methods to write numbers and bytes, and the Charset class convert text into byte arrays explicitly. A number of factors might influence your choice of character set:

- Performance: Can you reduce the number of conversions by transforming text into a character set that is used on the largest number of servers?
- Uniformity: Transfer all messages in the same character set.

- Richness: What character sets have all the code points that applications must use?
- Simplicity: Single-byte character sets are simpler to use than variable length and multibyte character sets.

See “Exchanging a formatted record with a non-JMS application” on page 179. for examples of converting messages exchanged with non-JMS applications.

Examples

Table of message types and conversion types

Table 31. Message types and conversion types

	Conversion type			
Message type	**Text**	**Numeric**	**Other**	**None**
JMSObjectMessage				getObject setObject
JMSTextMessage	getText setText			
JMSBytesMessage	readUTF writeUTF	readDouble readFloat readInt readLong readShort readUnsignedShort writeDouble writeFloat writeInt writeLong writeShort	readBoolean readObject writeBoolean writeObject	readByte readUnsignedByte readBytes readChar writeByte writeBytes writeChar
JMSStreamMessage	readString writeString	readDouble readFloat readInt readLong readShort writeDouble writeFloat writeInt writeLong writeShort	readBoolean readObject writeBoolean writeObject	readByte readBytes readChar writeByte writeBytes writeChar

Table 31. Message types and conversion types (continued)				
	Conversion type			
Message type	**Text**	**Numeric**	**Other**	**None**
JMSMapMessage	getString setString	getDouble getFloat getInt getLong getShort setDouble setFloat setInt setLong setShort	getBoolean getObject setBoolean setObject	getByte getBytes readChar setByte setBytes setChar

Calling data conversion from a C program

```
gmo.Options = MQGMO_WAIT         /* wait for new messages           */
            | MQGMO_NO_SYNCPOINT /* no transaction                 */
            | MQGMO_CONVERT;     /* convert if necessary            */

     while (CompCode != MQCC_FAILED) {
     buflen = sizeof(buffer) - 1; /* buffer size available for GET   */
     memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
     memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
     md.Encoding       = MQENC_NATIVE;
     md.CodedCharSetId = MQCCSI_Q_MGR;

           MQGET(Hcon,          /* connection handle               */
           Hobj,                /* object handle                   */
           &md,                 /* message descriptor              */
           &gmo,                /* get message options             */
           buflen,              /* buffer length                   */
           buffer,              /* message buffer                  */
           &messlen,            /* message length                  */
           &CompCode,           /* completion code                 */
           &Reason);            /* reason code                     */
```

Figure 13. Code snippet from `amqsget0.c`

Sending and receiving text in a JMSBytesMessage

The code in Figure 14 on page 164 sends a string in a BytesMessage. For simplicity, the example sends a single string, for which a `JMSTextMessage` is more appropriate. To receive a text string in bytes message containing a mixture of types, you must know the length of the string in bytes, called *TEXT_LENGTH* in Figure 15 on page 165. Even for a string with a fixed number of characters, the length of the byte representation might be longer.

```
BytesMessage bytes = session.createBytesMessage();
String codePage = CCSID.getCodepage(((MQDestination) destination)
                        .getIntProperty(WMQConstants.WMQ_CCSID));
bytes.writeBytes("In the destination code page".getBytes(codePage));
producer.send(bytes);
```

Figure 14. Sending a `String` in a `JMSBytesMessage`

```
BytesMessage message = (BytesMessage)consumer.receive();
int TEXT_LENGTH = new Long(message.getBodyLength())).intValue();
byte[] textBytes = new byte[TEXT_LENGTH];
message.readBytes(textBytes, TEXT_LENGTH);
String codePage = message.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET);
String textString = new String(textBytes, codePage);
```

Figure 15. Receiving a `String` from a `JMSBytesMessage`

Related concepts

JMS client message conversion and encoding
The methods you use to do JMS client message conversion and encoding are listed, with code examples of each type of conversion.

Queue manager data conversion
Queue manager data conversion has always been available to non-JMS applications receiving messages from JMS clients. JMS clients receiving messages also use queue manager data conversion, which is optional.

Related tasks

Exchanging a formatted record with a non-JMS application
Follow the steps suggested in this task to design and build a data conversion exit, and a JMS client application that can exchange messages with a non-JMS application using `JMSBytesMessage`. The exchange of a formatted message with a non-JMS application can take place with or without calling a data conversion exit.

Related reference

JMS message types and conversion
The choice of message type affects your approach to message conversion. The interaction of message conversion and message type is described for the JMS message types, `JMSObjectMessage`, `JMSTextMessage`, `JMSMapMessage`, `JMSStreamMessage`, and `JMSBytesMessage`.

JMS message types and conversion
The choice of message type affects your approach to message conversion. The interaction of message conversion and message type is described for the JMS message types, `JMSObjectMessage`, `JMSTextMessage`, `JMSMapMessage`, `JMSStreamMessage`, and `JMSBytesMessage`.

JMSObjectMessage

`JMSObjectMessage` contains one object, and any objects that it references, serialized into a byte stream by the JVM. Text is serialized into UTF-8, and limited to strings or character arrays of no more than 65534 bytes. An advantage of `JMSObjectMessage` is that applications are not involved in any data conversion issues as long as they use only the methods and attributes of the object. `JMSObjectMessage` provides data conversion for complex objects without the application programmer considering how to encode an object in a message. The disadvantage of using `JMSObjectMessage` is it can be exchanged only with other JMS applications. By choosing one of the other JMS message types, it is possible to exchange JMS messages with non-JMS applications.

"Sending and receiving a JMSObjectMessage" on page 168 shows a `String` object being exchanged in a message.

A JMS client application can receive a `JMSObjectMessage` only in a message that has a JMS-style body. The destination must specify a JMS style body.

JMSTextMessage

`JMSTextMessage` contains a single text string. When a text message is sent, the text `Format` is set to `"MQSTR   "`, `WMQConstants.MQFMT_STRING`. The `CodedCharacterSetId` of the text is set to the coded character set identifier defined for its destination. The text is encoded into the

CodedCharacterSetId by IBM MQ. The CodedCharacterSetId and Format fields are either set in the message descriptor, MQMD, or into the JMS fields in an MQRFH2. If the message is defined as having an WMQ_MESSAGE_BODY_MQ message body style, or the body style is unspecified, but the target destination is WMQ_TARGET_DEST_MQ, then the message descriptor fields are set. Otherwise the message has a JMS RFH2 and the fields are set in the fixed part of the MQRFH2.

An application can override the coded character set identifier defined for a destination. It must set the message property JMS_IBM_CHARACTER_SET to a coded character set identifier; see the example in "Sending and receiving a JMSTextmessage" on page 168.

When the JMS client calls the consumer.receive method queue manager conversion is optional. Queue manager conversion is enabled by setting the destination property WMQ_RECEIVE_CONVERSION to WMQ_RECEIVE_CONVERSION_QMGR. The queue manager converts the text message from the JMS_IBM_CHARACTER_SET specified for the message before transferring the message to the JMS client. The character set of the converted message is 1208, UTF-8, unless the destination has a different WMQ_RECEIVE_CCSID. The CodedCharacterSetId in the message that refers to the JMSTextMessage is updated to the target character set ID. The text is decoded from the target character set into Unicode by the getText method; see the example in "Sending and receiving a JMSTextmessage" on page 168.

A JMSTextMessage can be sent in an MQ-style message body, without a JMS MQRFH2 header. The value of the destination attributes, WMQ_MESSAGE_BODY and WMQ_TARGET_DEST determine the message body style, unless overridden by the application. The application can override the values set on the destination by calling destination.setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_MQ) or destination.setTargetClient(WMQConstants.WMQ_TARGET_DEST_MQ).

If you send a JMSTextMessage with an MQ style body by sending it to a destination with WMQ_MESSAGE_BODY set to WMQ_MESSAGE_BODY_MQ, you cannot receive it as a JMSTextMessage from the same destination. All messages received from a destination with WMQ_MESSAGE_BODY set to WMQ_MESSAGE_BODY_MQ are received as a JMSBytesMessage. If you try to receive the message as a JMSTextMessage it causes an exception, ClassCastException: com.ibm.jms.JMSBytesMessage cannot be cast to jakarta (or javax).jms.TextMessage.

Note: Text in a JMSBytesMessage is not converted by the JMS client. The client can only receive the text in the message as a byte array. If queue manager conversion is enabled, the text is converted by the queue manager, but the JMS client must still receive it as a byte array in a JMSBytesMessage.

It is generally better to use the WMQ_TARGET_DEST property to control whether a JMSTextMessage is sent with an MQ or JMS body style. You can then receive the message from a destination that has either WMQ_TARGET_DEST set to WMQ_TARGET_DEST_MQ or WMQ_TARGET_DEST_JMS. WMQ_TARGET_DEST has no effect on the receiver.

JMSMapMessage and JMSStreamMessage

These two JMS message types are similar. You can read and write primitive types to the messages using methods based on the DataInputStream and DataOutputStream interfaces; see "Table of message types and conversion types" on page 170. The details are described in "JMS client message conversion and encoding" on page 172. Each primitive is tagged; see "The JMS message body" on page 157.

Numeric data is read and written to the message encoded as XML text. No reference is made to the destination property, JMS_IBM_ENCODING. Text data is treated the same way as text in a JMSTextMessage. If you were to look at the message contents created by the example in Figure 20 on page 169, all the message data would be in EBCDIC as it was sent with a character set value of 37.

You can send multiple items in a JMSMapMessage or JMSStreamMessage.

You can retrieve the individual items of data by name from a JMSMapMessage, or by position from a JMSStreamMessage. Each item is decoded when a get or read method is called using the CodedCharacterSetId value stored in the message. If the method used to retrieve the item returns a different type to the type that was sent, the type is converted. If the type cannot be converted, an exception is thrown. See Class JMSStreamMessage for details. The example in "Sending data in

a JMSStreamMessage and JMSMapMessage" on page 169 illustrates type conversion, and getting the JMSMapMessage contents out of sequence.

The MQRFH2.format field for the JMSMapMessage and JMSStreamMessage is set to "MQSTR ". If the destination property WMQ_RECEIVE_CONVERSION is set to WMQ_RECEIVE_CONVERSION_QMGR, the message data is converted by the queue manager before being sent to the JMS client. The MQRFH2.CodedCharacterSetId of the message is the WMQ_RECEIVE_CCSID of the destination. The MQRFH2.Encoding is Native. If WMQ_RECEIVE_CONVERSION is WMQ_RECEIVE_CONVERSION_CLIENT_MSG the CodedCharacterSetId and Encoding of the MQRFH2 is the value set by the sender.

A JMS client application can receive a JMSMapMessage or JMSStreamMessage only in a message that has a JMS-style body, and from a destination that does not specify an MQ style body.

JMSBytesMessage

A JMSBytesMessage can contain multiple primitive types. You can read and write primitive types to the messages using methods based on the DataInputStream and DataOutputStream interfaces; see "Table of message types and conversion types" on page 170. The details are described in "JMS message types and conversion" on page 165.

The encoding of numeric data in the message is controlled by the value of JMS_IBM_ENCODING that is set before writing numeric data to the JMSBytesMessage. An application can override the default Native encoding defined for JMSBytesMessage by setting the message property JMS_IBM_ENCODING.

Text data can be read and written in UTF-8 using the readUTF and writeUTF, or in Unicode using the readChar and writeChar methods. There are no methods that use CodedCharacterSetId. Alternatively, the JMS client can encode and decode text into bytes using the Charset class. It transfers the bytes between the JVM and message without the IBM MQ classes for JMS performing any conversion; see "Sending and receiving text in a JMSBytesMessage" on page 169.

A JMSBytesMessage sent to an MQ application is typically sent in an MQ-style message body, without a JMS MQRFH2 header. If it is sent to a JMS application, the message body style is typically JMS. The value of the destination attributes, WMQ_MESSAGE_BODY and WMQ_TARGET_DEST determine the message body style, unless overridden by the application. The application can override the values set on the destination by calling destination.setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_MQ) or destination.setTargetClient(WMQConstants.WMQ_TARGET_DEST_MQ).

If you send a JMSBytesMessage with an MQ style body, you can receive the message from a destination that defines either an MQ or a JMS message body style. If you send a JMSBytesMessage with a JMS style body, then you must receive the message from a destination that defines a JMS message body style. If you do not, the MQRFH2 is treated as part of the user message data, which might not be what you are expecting.

Whether a message has an MQ or a JMS body style, the way it is received is not affected by setting WMQ_TARGET_DEST.

The message might be transformed later, by the queue manager, if a Format is supplied for the message data, and queue manager data conversion is enabled. Do not use the format field for anything other than specifying the format of the message data, or leave it blank, MQConstants.MQFMT_NONE

You can send multiple items in a JMSBytesMessage. Each numeric item is converted when the message is sent using the encoding defined for the message.

You can retrieve the individual items of data from JMSBytesMessage. Call read methods in the same order as the write methods were called to create the message. Each numeric item is converted when the message is called using the Encoding value stored in the message.

Unlike JMSMapMessage and JMSStreamMessage, JMSBytesMessage contains only data written by the application. No additional data is stored in the message data, such as the XML tags used to define the items in a JMSMapMessage and JMSStreamMessage. For this reason, use JMSBytesMessage to transfer messages formatted for other applications.

Converting between JMSBytesMessage and DataInputStream and DataOutputStream is useful in some applications. Code based on the example, “Reading and writing messages using DataInputStream and DataOutputStream” on page 169, is necessary to use the com.ibm.mq.header package with JMS.

Examples

Sending and receiving a JMSObjectMessage

```
ObjectMessage omo = session.createObjectMessage();
omo.setObject(new String("A string"));
producer.send(omo);
...
ObjectMessage omi = (ObjectMessage)consumer.receive();
System.out.println((String)omi.getObject());
...
A string
```

Figure 16. Sending and receiving a JMSObjectMessage

Sending and receiving a JMSTextmessage

A text message cannot contain text in different character sets. The example shows text in different character sets, sent in two different messages.

```
TextMessage tmo = session.createTextMessage();
tmo.setText("Sent in the character set defined for the destination");
producer.send(tmo);
```

Figure 17. Send text message in the character set defined by the destination

```
TextMessage tmo = session.createTextMessage();
tmo.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, 37);
tmo.setText("Sent in EBCDIC character set 37");
producer.send(tmo);
```

Figure 18. Send text message in ccsid 37

```
TextMessage tmi = (TextMessage)consumer.receive();
System.out.println(tmi.getText());
...
Sent in the character set defined for the destination
```

Figure 19. Receive text message

Sending data in a JMSStreamMessage and JMSMapMessage

```
StreamMessage smo = session.createStreamMessage();
smo.writeString("256");
smo.writeInt(512);
smo.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, 37);
producer.send(smo);
...
MapMessage mmo = session.createMapMessage();
mmo.setString("First", "256");
mmo.setInt("Second", 512);
mmo.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, 37);
producer.send(mmo);
...
StreamMessage smi = (StreamMessage)consumer.receive();
System.out.println("Stream: First as float "   + smi.readFloat() +
                         " Second as String " + smi.readString());
...
Stream: First as float: 256.0, Second as String: 512
...
MapMessage mmi = (MapMessage)consumer.receive();
System.out.println("Map: Second as String " + mmi.getString("Second") +
                       " First as double "  + mmi.getDouble("First"));
...
Map: Second as String: 512, First as double: 256.0
```

Figure 20. Send data in JMSStreamMessage and JMSMapMessage

Sending and receiving text in a JMSBytesMessage

The code in Figure 21 on page 169 sends a string in a BytesMessage. For simplicity, the example sends a single string, for which a JMSTextMessage is more appropriate. To receive a text string in bytes message containing a mixture of types, you must know the length of the string in bytes, called *TEXT_LENGTH* in Figure 22 on page 169. Even for a string with a fixed number of characters, the length of the byte representation might be longer.

```
BytesMessage bytes = session.createBytesMessage();
String codePage = CCSID.getCodepage(((MQDestination) destination)
                    .getIntProperty(WMQConstants.WMQ_CCSID));
bytes.writeBytes("In the destination code page".getBytes(codePage));
producer.send(bytes);
```

Figure 21. Sending a String in a JMSBytesMessage

```
BytesMessage message = (BytesMessage)consumer.receive();
int TEXT_LENGTH = new Long(message.getBodyLength())).intValue();
byte[] textBytes = new byte[TEXT_LENGTH];
message.readBytes(textBytes, TEXT_LENGTH);
String codePage = message.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET);
String textString = new String(textBytes, codePage);
```

Figure 22. Receiving a String from a JMSBytesMessage

Reading and writing messages using DataInputStream and DataOutputStream

The code in Figure 23 on page 170 creates a JMSBytesMessage using a DataOutputStream.

```
ByteArrayOutputStream bout = new ByteArrayOutputStream();
DataOutputStream dout = new DataOutputStream(bout);
BytesMessage messageOut = prod.session.createBytesMessage();
// messageOut.setIntProperty(WMQConstants.JMS_IBM_ENCODING,
//                           ((MQDestination) (prod.destination)).getIntProperty
//                           (WMQConstants.WMQ_ENCODING));
int ccsidOut = (((MQDestination)prod.destination).getIntProperty(WMQConstants.WMQ_CCSID));
String codePageOut = CCSID.getCodepage(ccsidOut);
dout.writeInt(ccsidOut);
dout.write(codePageOut.getBytes(codePageOut));
messageOut.writeBytes(bout.toByteArray());
producer.send(messageOut);
```

Figure 23. Send a JMSBytesMessage using a DataOutputStream

The statement that sets the JMS_IBM_ENCODING property is commented out. The statement is valid, if writing directly to a JMSBytesMessage, but has no effect when writing to DataOutputStream. Numbers that are written to the DataOutputStream are encoded in Native encoding. Setting JMS_IBM_ENCODING has no effect.

The code in Figure 24 on page 170 receives a JMSBytesMessage using a DataInputStream.

```
static final int ccsidIn_SIZE = (Integer.SIZE)/8;
...
connection.start();
BytesMessage messageIn = (BytesMessage) consumer.receive();
int messageLength = new Long(messageIn.getBodyLength()).intValue();
byte [] bin = new byte[messageLength];
messageIn.readBytes(bin, messageLength);
DataInputStream din = new DataInputStream(new ByteArrayInputStream(bin));
int ccsidIn = din.readInt();
byte [] codePageByte = new byte[messageLength - ccsidIn_SIZE];
din.read(codePageByte, 0, codePageByte.length);
System.out.println("CCSID " + ccsidIn + " code page " + new String(codePageByte,
                    messageIn.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET)));
```

Figure 24. Receive a JMSBytesMessage using a DataInputStream

The code page is printed out using the code page property of the input message data, JMS_IBM_CHARACTER_SET. On input JMS_IBM_CHARACTER_SET is a Java code page and not a numeric coded character set identifier.

Table of message types and conversion types

Table 32. Message types and conversion types

	Conversion type			
Message type	**Text**	**Numeric**	**Other**	**None**
JMSObjectMessage				getObject setObject
JMSTextMessage	getText setText			

Table 32. Message types and conversion types (continued)				
	Conversion type			
Message type	**Text**	**Numeric**	**Other**	**None**
JMSBytesMessage	readUTF writeUTF	readDouble readFloat readInt readLong readShort readUnsignedShort writeDouble writeFloat writeInt writeLong writeShort	readBoolean readObject writeBoolean writeObject	readByte readUnsignedByte readBytes readChar writeByte writeBytes writeChar
JMSStreamMessage	readString writeString	readDouble readFloat readInt readLong readShort writeDouble writeFloat writeInt writeLong writeShort	readBoolean readObject writeBoolean writeObject	readByte readBytes readChar writeByte writeBytes writeChar
JMSMapMessage	getString setString	getDouble getFloat getInt getLong getShort setDouble setFloat setInt setLong setShort	getBoolean getObject setBoolean setObject	getByte getBytes readChar setByte setBytes setChar

Related concepts

JMS message conversion approaches

A number of data conversion approaches are open to JMS application designers. These approaches are not exclusive; some applications are likely to use a combination of these approaches. If your application is exchanging only text or is exchanging messages only with other JMS applications, you do not normally consider data conversion. Data conversion is performed automatically for you, by IBM MQ.

JMS client message conversion and encoding

The methods you use to do JMS client message conversion and encoding are listed, with code examples of each type of conversion.

Queue manager data conversion

Queue manager data conversion has always been available to non-JMS applications receiving messages from JMS clients. JMS clients receiving messages also use queue manager data conversion, which is optional.

Related tasks

Exchanging a formatted record with a non-JMS application

Follow the steps suggested in this task to design and build a data conversion exit, and a JMS client application that can exchange messages with a non-JMS application using JMSBytesMessage. The exchange of a formatted message with a non-JMS application can take place with or without calling a data conversion exit.

JMS client message conversion and encoding

The methods you use to do JMS client message conversion and encoding are listed, with code examples of each type of conversion.

Conversion and encoding occur when Java primitives or objects are read or written to and from JMS messages. The conversion is called JMS client data conversion to distinguish it from queue manager data conversion and application data conversion. The conversion takes place strictly when data is read from or written to a JMS message. Text is converted to and from the internal 16 bit Unicode representation [3] to the character set used for text in messages. Numeric data is converted to and Java primitive numeric types to the encoding defined for the message. Whether conversion is performed, and what type of conversion is performed, depends on the JMS message type and the read or write operation.

Table 33 on page 172 categorizes the read and write methods for different JMS message types by the type of conversion performed. The conversions types are described in the text following the table.

Table 33. Message types and conversion types

	Conversion type			
Message type	**Text**	**Numeric**	**Other**	**None**
JMSObjectMessage				getObject setObject
JMSTextMessage	getText setText			
JMSBytesMessage	readUTF writeUTF	readDouble readFloat readInt readLong readShort readUnsignedShort writeDouble writeFloat writeInt writeLong writeShort	readBoolean readObject writeBoolean writeObject	readByte readUnsignedByte readBytes readChar writeByte writeBytes writeChar

[3] Some Unicode representation requires more than 16 bits. See a Java SE reference.

Table 33. Message types and conversion types (continued)				
	Conversion type			
Message type	**Text**	**Numeric**	**Other**	**None**
JMSStreamMessage	readString writeString	readDouble readFloat readInt readLong readShort writeDouble writeFloat writeInt writeLong writeShort	readBoolean readObject writeBoolean writeObject	readByte readBytes readChar writeByte writeBytes writeChar
JMSMapMessage	getString setString	getDouble getFloat getInt getLong getShort setDouble setFloat setInt setLong setShort	getBoolean getObject setBoolean setObject	getByte getBytes readChar setByte setBytes setChar

Text

The default CodedCharacterSetId for a destination is 1208, UTF-8. By default, text is converted from Unicode and sent as a UTF-8 text string. On receive, the text is converted from the coded character set in the message received by the client, into Unicode.

The setText and writeString methods convert text from Unicode into the character set defined for the destination. An application can override the destination character set by setting the message property JMS_IBM_CHARACTER_SET. JMS_IBM_CHARACTER_SET, when sending a message must be a numeric coded character set identifier [4] .

The code snippets in "Sending and receiving a JMSTextmessage" on page 175 send two messages. One is sent in the character set defined for the destination and the other in character set 37, defined by the application.

The getText and readString methods convert the text in the message from the character set defined in the message into Unicode. The methods use the code page defined in the message property, JMS_IBM_CHARACTER_SET. The code page is mapped from MQRFH2.CodedCharacterSetId unless the message is an MQ-type message and has no MQRFH2. If the message is a MQ-type message, with no MQRFH2, the code page is mapped from MQMD.CodedCharacterSetId.

The code snippet in Figure 29 on page 176 receives the message that was sent to the destination. The text in the message is converted from code page IBM037 back into Unicode.

Note: A simple way to check that the text is converted to coded character set 37 is to use IBM MQ Explorer. Browse the queue and show the properties of the message before it is retrieved.

Contrast the code snippet in Figure 28 on page 176 with the incorrect code snippet in Figure 25 on page 174. In the incorrect snippet the text string is converted twice, once by the application, and again by IBM MQ.

[4] When receiving a message JMS_IBM_CHARACTER_SET is a Java Charset code page name.

```
TextMessage tmo = session.createTextMessage();
tmo.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, 37);
tmo.setText(new String("Sent in EBCDIC character set 37".getBytes(CCSID.getCodepage(37))));
producer.send(tmo);
```

Figure 25. Incorrect code page conversion

The writeUTF method converts text from Unicode to 1208, UTF-8. The text string is prefaced with a 2 byte length. The maximum length of the text string is 65534 bytes. The readUTF method reads an item in a message written by the writeUTF method. It reads exactly the number of bytes written by the writeUTF method.

Numeric

The default numeric encoding for a destination is Native. The Native encoding constant for Java has the value 273, x'00000111', which is the same for all platforms. On receive, the numbers in the message are correctly transformed into numeric Java primitives. The transformation uses the encoding defined in the message and the type returned by the read method.

The send method converts numbers that are added to a message by the set and write into the numeric encoding defined for the destination. The destination encoding can be overridden for a message by an application setting the message property, JMS_IBM_ENCODING ; for example:

```
message.setIntProperty(WMQConstants.JMS_IBM_ENCODING,
WMQConstants.WMQ_ENCODING_INTEGER_REVERSED);
```

The get and read numeric methods convert numbers in the message from the numeric encoding defined in the message. They convert the numbers to the type that is specified by the read or get method; see The ENCODING property. The methods use the encoding defined in JMS_IBM_ENCODING. The encoding is mapped from MQRFH2.Encoding unless the message is an MQ-type message and has no MQRFH2. If the message is a MQ-type message, with no MQRFH2, then the methods use the encoding defined in MQMD.Encoding.

The example in Figure 30 on page 176 shows an application encoding a number in the destination format and sending it in a JMSStreamMessage. Compare the example in Figure 30 on page 176 to the example in Figure 31 on page 176. The difference is that JMS_IBM_ENCODING must be set in a JMSBytesMessage.

Note: A simple way to check that the number is encoded correctly is to use IBM MQ Explorer. Browse the queue and show the properties of the message before it is consumed.

Other

The boolean methods encode true and false as x'01' and x'00' in a JMSByteMessage, JMSStreamMessage, and JMSMapMessage.

The UTF methods encode and decode Unicode into UTF-8 text strings. The strings are limited to less than 65536 characters, and are preceded by the 2 byte length field.

The Object methods wrap primitive types as objects. Numeric and text types are encoded or converted as if the primitive types had been read or written using the numeric and text methods.

None

The readByte, readBytes, readUnsignedByte, writeByte, and writeBytes methods get or put single bytes, or arrays of bytes, between the application and the message without conversion. The readChar and writeChar methods get and put 2 byte Unicode characters between the application and the message without conversion.

Using the readBytes and writeBytes methods, the application can perform its own code point conversion, as in “Sending and receiving text in a JMSBytesMessage” on page 177.

IBM MQ does not perform any code page conversion in the client as the message is a JMSBytesMessage, and because the readBytes and writeBytes methods are used. Nonetheless, if the bytes represent text, make sure that code page used by the application matches the coded character set of the destination. The message might be converted again by a queue manager conversion exit. Another possibility is that the receiving JMS client program might follow the convention of converting any byte arrays representing text in the message into strings or characters using the JMS_IBM_CHARACTER_SET property in the message.

In this example the client uses the destination coded character set for its conversion:

```
bytes.writeBytes("In the destination code page".getBytes(
CCSID.getCodepage(((MQDestination) destination)
.getIntProperty(WMQConstants.WMQ_CCSID))));
```

Alternatively, the client might have chosen a code page and then set the corresponding coded character set in the JMS_IBM_CHARACTER_SET property of the message. The IBM MQ classes for Java use JMS_IBM_CHARACTER_SET to set the CodedCharacterSetId field in the JMS properties in the MQRFH2, or in the message descriptor, MQMD:

```
String codePage = CCSID.getCodepage(37);
message.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, codePage);
5
```

If a byte array is written into a JMSStringMessage or JMSMapMessage, IBM MQ classes for JMS does not perform data conversion, as the bytes are typed as hexadecimal data not as text in the JMSStringMessage and JMSMapMessage.

If the bytes represent characters in your application, you must take into account what code points to read and write to the message. The code in Figure 26 on page 175 follows the convention of using the destination coded character set. If you create the string using the default character set for the JVM, the byte contents depend on the platform. A JVM on Windows typically has a default Charset of windows-1252, and AIX and Linux has UTF-8. For interchange between Windows, and AIX and Linux, you have to select an explicit code page for exchanging text as bytes.

```
StreamMessage smo = producer.session.createStreamMessage();
smo.writeBytes("123".getBytes(CCSID.getCodepage(((MQDestination) destination)
.getIntProperty(WMQConstants.WMQ_CCSID))));
```

Figure 26. Writing bytes representing a string in a JMSStreamMessage using the destination character set

Examples

Sending and receiving a JMSTextmessage

A text message cannot contain text in different character sets. The example shows text in different character sets, sent in two different messages.

[5] SetStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET, codePage) currently accepts only numeric character set identifiers.

```
TextMessage tmo = session.createTextMessage();
tmo.setText("Sent in the character set defined for the destination");
producer.send(tmo);
```

Figure 27. Send text message in the character set defined by the destination

```
TextMessage tmo = session.createTextMessage();
tmo.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, 37);
tmo.setText("Sent in EBCDIC character set 37");
producer.send(tmo);
```

Figure 28. Send text message in `ccsid 37`

```
TextMessage tmi = (TextMessage)consumer.receive();
System.out.println(tmi.getText());
...
Sent in the character set defined for the destination
```

Figure 29. Receive text message

Encoding examples

Examples showing a number being sent in the encoding defines for a destination. Notice that you must set the `JMS_IBM_ENCODING` property of a `JMSBytesMessage` to the value specified for the destination.

```
StreamMessage smo = session.createStreamMessage();
smo.writeInt(256);
producer.send(smo);
...
StreamMessage smi = (StreamMessage)consumer.receive();
System.out.println(smi.readInt());
...
256
```

Figure 30. Sending a number using the destination encoding in a `JMSStreamMessage`

```
BytesMessage bmo = session.createBytesMessage();
bmo.writeInt(256);
int encoding = ((MQDestination) (destination)).getIntProperty
    (WMQConstants.WMQ_ENCODING)
bmo.setIntProperty(WMQConstants.JMS_IBM_ENCODING, encoding);
producer.send(bmo);
...
BytesMessage bmi = (BytesMessage)consumer.receive();
System.out.println(bmi.readInt());
...
256
```

Figure 31. Sending a number using the destination encoding in a `JMSBytesMessage`

Sending and receiving text in a JMSBytesMessage

The code in Figure 32 on page 177 sends a string in a BytesMessage. For simplicity, the example sends a single string, for which a JMSTextMessage is more appropriate. To receive a text string in bytes message containing a mixture of types, you must know the length of the string in bytes, called *TEXT_LENGTH* in Figure 33 on page 177. Even for a string with a fixed number of characters, the length of the byte representation might be longer.

```
BytesMessage bytes = session.createBytesMessage();
String codePage = CCSID.getCodepage(((MQDestination) destination)
                    .getIntProperty(WMQConstants.WMQ_CCSID));
bytes.writeBytes("In the destination code page".getBytes(codePage));
producer.send(bytes);
```

Figure 32. Sending a String in a JMSBytesMessage

```
BytesMessage message = (BytesMessage)consumer.receive();
int TEXT_LENGTH = new Long(message.getBodyLength())).intValue();
byte[] textBytes = new byte[TEXT_LENGTH];
message.readBytes(textBytes, TEXT_LENGTH);
String codePage = message.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET);
String textString = new String(textBytes, codePage);
```

Figure 33. Receiving a String from a JMSBytesMessage

Related concepts

JMS message conversion approaches
A number of data conversion approaches are open to JMS application designers. These approaches are not exclusive; some applications are likely to use a combination of these approaches. If your application is exchanging only text or is exchanging messages only with other JMS applications, you do not normally consider data conversion. Data conversion is performed automatically for you, by IBM MQ.

Queue manager data conversion
Queue manager data conversion has always been available to non-JMS applications receiving messages from JMS clients. JMS clients receiving messages also use queue manager data conversion, which is optional.

Related tasks

Exchanging a formatted record with a non-JMS application
Follow the steps suggested in this task to design and build a data conversion exit, and a JMS client application that can exchange messages with a non-JMS application using JMSBytesMessage. The exchange of a formatted message with a non-JMS application can take place with or without calling a data conversion exit.

Related reference

JMS message types and conversion
The choice of message type affects your approach to message conversion. The interaction of message conversion and message type is described for the JMS message types, JMSObjectMessage, JMSTextMessage, JMSMapMessage, JMSStreamMessage, and JMSBytesMessage.

Queue manager data conversion
Queue manager data conversion has always been available to non-JMS applications receiving messages from JMS clients. JMS clients receiving messages also use queue manager data conversion, which is optional.

The queue manager can convert character and numeric data in message data using the values of CodedCharacterSetId, Encoding, and Format set for the message data. For non-JMS applications the conversion capability has always been available by setting the GetMessageOption, GMO_CONVERT.

The queue manager is able to convert messages that are sent to JMS clients. Queue manager conversion is controlled by setting the destination property, `WMQ_RECEIVE_CONVERSION`, to `WMQ_RECEIVE_CONVERSION_QMGR`, or `WMQ_RECEIVE_CONVERSION_CLIENT_MSG`. The application can change the destination setting:

```
((MQDestination)destination).setIntProperty(
              WMQConstants.WMQ_RECEIVE_CONVERSION,
              WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
```

Or,

```
((MQDestination)destination).setReceiveConversion
              (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
```

Figure 34. Enable queue manager data conversion

Queue manager data conversion for a JMS client takes place when the client calls a `consumer.receive` method. Text data is transformed into UTF-8 (1208) by default. Subsequent read and get methods decode text in the received data from UTF-8, creating Java text primitives in their internal Unicode encoding. UTF-8 is not the only target character set from queue manager data conversion. You can choose a different CCSID by setting the `WMQ_RECEIVE_CCSID` destination property.

An application can also change the destination setting, for example setting it to 437, DOS-US:

```
((MQDestination)destination).setIntProperty
(WMQConstants.WMQ_RECEIVE_CCSID, 437);
```

Or,

```
((MQDestination)destination).setReceiveCCSID(437);
```

Figure 35. Set target coded character set for queue manager conversion

The reason for changing `WMQ_RECEIVE_CCSID` is specialized; the chosen CCSID makes no difference to the text objects created in the JVM. However, some JVMs, on some platforms, might not be able to handle conversion from the CCSID of text in the message into Unicode. The option gives you a choice of CCSID for any text delivered to the client in the message. Some JMS client platforms have had problems with message text being delivered in UTF-8.

The JMS code is equivalent to the bold text in the C code in Figure 36 on page 179,

```
gmo.Options = MQGMO_WAIT         /* wait for new messages           */
            | MQGMO_NO_SYNCPOINT /* no transaction                  */
            | MQGMO_CONVERT;     /* convert if necessary            */

     while (CompCode != MQCC_FAILED) {
     buflen = sizeof(buffer) - 1; /* buffer size available for GET   */
     memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
     memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
     md.Encoding       = MQENC_NATIVE;
     md.CodedCharSetId = MQCCSI_Q_MGR;

           MQGET(Hcon,          /* connection handle               */
           Hobj,                /* object handle                   */
           &md,                 /* message descriptor              */
           &gmo,                /* get message options             */
           buflen,              /* buffer length                   */
           buffer,              /* message buffer                  */
           &messlen,            /* message length                  */
           &CompCode,           /* completion code                 */
           &Reason);            /* reason code                     */
```

Figure 36. Code snippet from `amqsget0.c`

Note:

Queue manager conversion is only performed on the message data that has a known IBM MQ format. MQSTR, or MQCIH are examples of known formats that are predefined. A known format can also be user-defined format, as long as you have supplied a data-conversion exit.

Messages constructed as JMSTextMessage, JMSMapMessage and JMSStreamMessage, have a MQSTR format, and can be converted by the queue manager.

Related concepts

JMS message conversion approaches
A number of data conversion approaches are open to JMS application designers. These approaches are not exclusive; some applications are likely to use a combination of these approaches. If your application is exchanging only text or is exchanging messages only with other JMS applications, you do not normally consider data conversion. Data conversion is performed automatically for you, by IBM MQ.

JMS client message conversion and encoding
The methods you use to do JMS client message conversion and encoding are listed, with code examples of each type of conversion.

"Invoking the data-conversion exit" on page 948
A data-conversion exit is a user-written exit that receives control during the processing of an MQGET call.

Related tasks

Exchanging a formatted record with a non-JMS application
Follow the steps suggested in this task to design and build a data conversion exit, and a JMS client application that can exchange messages with a non-JMS application using JMSBytesMessage. The exchange of a formatted message with a non-JMS application can take place with or without calling a data conversion exit.

Related reference

JMS message types and conversion
The choice of message type affects your approach to message conversion. The interaction of message conversion and message type is described for the JMS message types, JMSObjectMessage, JMSTextMessage, JMSMapMessage, JMSStreamMessage, and JMSBytesMessage.

Exchanging a formatted record with a non-JMS application
Follow the steps suggested in this task to design and build a data conversion exit, and a JMS client application that can exchange messages with a non-JMS application using JMSBytesMessage. The

exchange of a formatted message with a non-JMS application can take place with or without calling a data conversion exit.

Before you begin

You might be able to design a simpler solution to exchanging messages with a non-JMS application using a `JMSTextMessage`. Eliminate that possibility before following the steps in this task.

About this task

A JMS client is easier to write if it is not involved in the details of formatting JMS messages exchanged with other JMS clients. As long as the message type is `JMSTextMessage`, `JMSMapMessage`, `JMSStreamMessage`, or `JMSObjectMessage`, IBM MQ looks after the details of formatting the message. IBM MQ deals with differences in code pages and numeric encoding on different platforms.

You can use these message types to exchange messages with non-JMS applications. To do so, you must understand how these messages are constructed by IBM MQ classes for JMS. You might be able to modify the non-JMS application to interpret the messages; see "Mapping JMS messages onto IBM MQ messages" on page 142.

An advantage of using one of these message types is the JMS client programming does not depend on the type of application that it is exchanging messages with. A disadvantage is that it might require a modification to another program, and you might not be able to change the other program.

An alternative approach is to write a JMS client application that can deal with existing message formats. Often existing messages are fixed format and contain a mixture of unformatted data, text, and numbers. Use the steps in this task, and the example JMS client in "Writing classes to encapsulate a record layout in a JMSBytesMessage" on page 183, as a starting point for building a JMS client that can exchange formatted records with non-JMS applications.

Procedure

1. Define the record layout, or use one of the predefined IBM MQ header classes.

   For handling predefined IBM MQ headers, see Handling IBM MQ message headers.

   Figure 37 on page 181 is an example of a user defined, fixed-length record layout, which can be processed by the data conversion utility.
2. Create the data conversion exit.

   Follow the instructions in Writing a data-conversion exit program to write a data conversion exit.

   To try out the example in "Writing classes to encapsulate a record layout in a JMSBytesMessage" on page 183, name the data conversion exit `MYRECORD`.
3. Write Java classes to encapsulate the record layout, and sending and receiving record. Two approaches you might take are:

   - Write a class to that reads and writes the `JMSBytesMessage` that contains the record; see "Writing classes to encapsulate a record layout in a JMSBytesMessage" on page 183.
   - Write a class extending `com.ibm.mq.header.Header` to define the data structure of the record; see Creating classes for new header types.
4. Decide what coded character set to exchange messages in.

   See Choosing an approach to message conversion: receiver makes good.
5. Configure the destination to exchange MQ-type messages, without a JMS `MQRFH2` header.

   Both the sending and receiving destination must be configured to exchange MQ-type messages. You can use the same destination for both sending and receiving.

   The application can override the destination message body property:

   ```
   ((MQDestination)destination).setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_MQ);
   ```

The example in "Writing classes to encapsulate a record layout in a JMSBytesMessage" on page 183 overrides the destination message body property, ensuring an MQ-style message is sent.

6. Test the solution with JMS and non-JMS applications

   Useful tools to test a data conversion exit are:

   - The `amqsgetc0.c` sample program is useful to test receiving a message sent by a JMS client. See the suggested modifications to use the example header, `RECORD.h`, in Figure 38 on page 182. With the modifications, `amqsgetc0.c` receives a message sent by the example JMS client, `TryMyRecord.java` ; see "Writing classes to encapsulate a record layout in a JMSBytesMessage" on page 183.
   - The sample IBM MQ browse program, `amqsbcg0.c`, is useful to inspect the contents of the message header, the JMS header, `MQRFH2`, and the message contents.
   - The **rfhutil** program, previously available in SupportPac IH03, allows test messages to be captured and stored in files, and then used to drive Message Flows. Output messages can also be read and displayed in a variety of formats. The formats include two types of XML as well as matching against a COBOL copybook. The data can be in EBCDIC or ASCII. An RFH2 header can be added to the message before the message is sent.

   If you try to receive messages using the modified `amqsgetc0.c` sample program, and get an error with reason code `2080`, check whether the message has an `MQRFH2`. The modifications assume that the message has been sent to a destination that specifies no `MQRFH2`.

Examples

```
struct RECORD { MQCHAR StrucID[4];
                MQLONG Version;
                               MQLONG StructLength;
                               MQLONG Encoding;
                               MQLONG CodeCharSetId;
                               MQCHAR Format[8];
                               MQLONG Flags;
                               MQCHAR RecordData[32];
        };
```

Figure 37. RECORD.h

- Declare the RECORD.h data structure

```
struct tagRECORD {
   MQCHAR4    StrucId;
   MQLONG     Version;
   MQLONG     StrucLength;
   MQLONG     Encoding;
   MQLONG     CCSID;
   MQCHAR8    Format;
   MQLONG     Flags;
   MQCHAR32   RecordData;
 };
typedef struct tagRECORD RECORD;
   typedef RECORD MQPOINTER PRECORD;
   RECORD record;
   PRECORD pRecord = &(record);
```

- Modify the MQGET call to use RECORD,

  1. Before modification:

```
    MQGET(Hcon,            /* connection handle     */
          Hobj,            /* object handle         */
          &md,             /* message descriptor    */
          &gmo,            /* get message options   */
          buflen,          /* buffer length         */
          buffer,          /* message buffer        */
          &messlen,        /* message length        */
          &CompCode,       /* completion code       */
          &Reason);        /* reason code           */
```

  2. After modification:

```
    MQGET(Hcon,            /* connection handle     */
          Hobj,            /* object handle         */
          &md,             /* message descriptor    */
          &gmo,            /* get message options   */
          sizeof(RECORD),  /* buffer length         */
          pRecord,         /* message buffer        */
          &messlen,        /* message length        */
          &CompCode,       /* completion code       */
          &Reason);        /* reason code           */
```

- Change the print statement,

  1. From:

```
buffer[messlen] = '\0';            /* add terminator          */
printf("message <%s>\n", buffer);
```

  2. To:

```
/* buffer[messlen] = '\0';            add terminator          */
printf("ccsid <%d>, flags <%d>, message <%32.32s>\n \0",
        md.CodedCharSetId, record.Flags, record.RecordData);
```

Figure 38. Modify amqsget0.c

Related concepts

JMS message conversion approaches

A number of data conversion approaches are open to JMS application designers. These approaches are not exclusive; some applications are likely to use a combination of these approaches. If your application is exchanging only text or is exchanging messages only with other JMS applications, you do not normally consider data conversion. Data conversion is performed automatically for you, by IBM MQ.

JMS client message conversion and encoding

The methods you use to do JMS client message conversion and encoding are listed, with code examples of each type of conversion.

Queue manager data conversion
Queue manager data conversion has always been available to non-JMS applications receiving messages from JMS clients. JMS clients receiving messages also use queue manager data conversion, which is optional.

Utility for creating conversion-exit code

Related reference

JMS message types and conversion
The choice of message type affects your approach to message conversion. The interaction of message conversion and message type is described for the JMS message types, `JMSObjectMessage`, `JMSTextMessage`, `JMSMapMessage`, `JMSStreamMessage`, and `JMSBytesMessage`.

Writing classes to encapsulate a record layout in a `JMSBytesMessage`
The purpose of this task is to explore, by example, how to combine data conversion and a fixed record layout in a `JMSBytesMessage`. In the task, you create some Java classes to exchange an example record structure in a `JMSBytesMessage`. You can modify the example to write classes to exchange other record structures.

A `JMSBytesMessage` is the best choice of JMS message type to exchange mixed data type records with non-JMS programs. It has no additional data inserted into the message body by the JMS provider. It is therefore the best choice of message type to use if a JMS client program interoperates with an existing IBM MQ program. The main challenge in using a `JMSBytesMessage` comes with matching the encoding and character set expected by the other program. A solution is to create a class that encapsulates the record. A class that encapsulates reading and writing a `JMSBytesMessage`, for a specific record type, makes it easier to send and receive fixed-format records in a JMS program. By capturing the generic aspects of the interface in an abstract class, much of the solution can be reused for different record formats. Different record formats can be implemented in classes that extend the abstract generic class.

An alternative approach is to extend the `com.ibm.mq.headers.Header` class. The `Header` class has methods, such as `addMQLONG`, to build a record format in a more declarative way. A disadvantage of using the `Header` class is getting and setting attributes uses a more complicated interpretative interface. Both approaches result in much the same amount of application code.

A `JMSBytesMessage` can encapsulate only a single format, in addition to an `MQRFH2`, in one message, unless each record uses the same format, coded character set, and encoding. The format, encoding, and character set of a `JMSBytesMessage` are properties of all of the message following the `MQRFH2`. The example is written on the assumption that a `JMSBytesMessage` contains only one user record.

Before you begin

1. Your skill level: you must be familiar with Java programming and JMS. No instructions are provided about setting up the Java development environment. It is advantageous to have written a program to exchange a `JMSTextMessage`, `JMSStreamMessage`, or `JMSMapMessage`. You can then see the differences in exchanging a message using a `JMSBytesMessage`.
2. The example requires IBM WebSphere MQ 7.0.
3. The example was created using the Java perspective of the Eclipse workbench. It requires JRE 6.0 or higher. You can use the Java perspective in IBM MQ Explorer to develop and run the Java classes. Alternatively, use your own Java development environment.
4. Using IBM MQ Explorer makes setting up the test environment, and debugging, simpler than using command-line utilities.

About this task

You are guided through creating two classes: `RECORD` and `MyRecord`. Together these two classes encapsulate a fixed-format record. They have methods to get and set attributes. The get method reads the record from a `JMSBytesMessage` and the put method writes a record to a `JMSBytesMessage`.

The purpose of the task is not to create a production quality class that you can reuse. You might choose to use the examples in the task to get started on your own classes. The purpose of the task is to provide you with guidance notes, primarily about using character sets, formats, and encoding, when using a JMSBytesMessage. Each step in creating the classes is explained, and aspects of using JMSBytesMessage, which are sometimes overlooked, are described.

The RECORD class is abstract and defines some common fields for a user record. The common fields are modeled on the standard IBM MQ header layout of having an eye catcher, a version, and a length field. The encoding, character set, and format fields, found in many IBM MQ headers, are omitted. Another header cannot follow a user-defined format. The MyRecord class, which extends the RECORD class, does so by literally extending the record with additional user fields. A JMSBytesMessage, created by the classes, can be processed by the queue manager data conversion exit.

"Classes used to run example" on page 190 includes a full listing of RECORD and MyRecord. It also includes listings of the extra "scaffolding" classes to test the RECORD and MyRecord. The extra classes are:

TryMyRecord
The main program to test RECORD and MyRecord.

EndPoint
An abstract class that encapsulates the JMS connection, destination, and session in a single class. Its interface just meets the needs of testing the RECORD and MyRecord classes. It is not an established design pattern for writing JMS applications.

Note: The Endpoint class includes this line of code after creating a destination:

```
((MQDestination)destination).setReceiveConversion
                            (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
```

In V7.0, from V7.0.1.5, it is necessary to turn on queue manager conversion. It is disabled by default. In V7.0, up to V7.0.1.4 queue manager conversion is enabled by default, and this line of code causes an error.

MyProducer and MyConsumer
Classes that extend EndPoint, and create a MessageConsumer and MessageProducer, connected and ready to accept requests.

Together all the classes make up a complete application you can build and experiment with, to understand how to use data conversion in a JMSBytesMessage.

Procedure

1. Create an abstract class to encapsulate the standard fields in an IBM MQ header, with a default constructor. Later, you extend the class to tailor the header to your requirements.

```
public abstract class RECORD implements Serializable {
   private static final long serialVersionUID = -1616617232750561712L;
   protected final static int UTF8 = 1208;
   protected final static int MQLONG_LENGTH = 4;
   protected final static int RECORD_STRUCT_ID_LENGTH = 4;
   protected final static int RECORD_VERSION_1 = 1;
   protected final String RECORD_STRUCT_ID = "BLNK";
   protected final String RECORD_TYPE = "BLANK   ";
   private String structID = RECORD_STRUCT_ID;
   private int version = RECORD_VERSION_1;
   private int structLength = RECORD_STRUCT_ID_LENGTH + MQLONG_LENGTH * 2;
   private int headerEncoding = WMQConstants.WMQ_ENCODING_NATIVE;
   private String headerCharset = "UTF-8";
   private String headerFormat = RECORD_TYPE;

   public RECORD() {
       super();
   }
```

Note:

a. The attributes, `structID` to `nextFormat`, are listed in the order they are laid out in a standard IBM MQ message header.

b. The attributes, `format`, `messageEncoding`, and `messageCharset`, describe the header itself, and are not part of the header.

c. You must decide whether to store the coded character set identifier or character set of the record. Java uses character sets and IBM MQ messages use coded character set identifiers. The example code uses character sets.

d. `int` is serialized to `MQLONG` by IBM MQ. `MQLONG` is 4 bytes.

2. Create the getters and setters for the private attributes.

a) Create or generate the getters:

```
     public String getHeaderFormat() { return headerFormat; }
    public int getHeaderEncoding() { return headerEncoding;    }
    public String getMessageCharset() { return headerCharset; }
    public int getMessageEncoding() { return headerEncoding; }
    public String getStructID() { return structID; }
    public int getStructLength() { return structLength; }
    public int getVersion() { return version; }
```

b) Create or generate the setters:

```
     public void setHeaderCharset(String charset) {
        this.headerCharset = charset; }
    public void setHeaderEncoding(int encoding) {
        this.headerEncoding = encoding; }
    public void setHeaderFormat(String headerFormat) {
        this.headerFormat = headerFormat; }
    public void setStructID(String structID) {
        this.structID = structID; }
    public void setStructLength(int structLength) {
        this.structLength = structLength; }
    public void setVersion(int version) {
        this.version = version; }
}
```

3. Create a constructor to create a RECORD instance from a `JMSBytesMessage`.

```
  public RECORD(BytesMessage message) throws JMSException, IOException,
         MQDataException {
     super();
     setHeaderCharset(message.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET));
     setHeaderEncoding(message.getIntProperty(WMQConstants.JMS_IBM_ENCODING));
     byte[] structID = new byte[RECORD_STRUCT_ID_LENGTH];
     message.readBytes(structID, RECORD_STRUCT_ID_LENGTH);
     setStructID(new String(structID, getMessageCharset()));
     setVersion(message.readInt());
     setStructLength(message.readInt());
  }
```

Note:

a. The `messageCharset` and `messageEncoding`, are captured from the message properties, as they override the values set for the destination. `format` is not updated. The example does no error checking. If the `Record(BytesMessage)` constructor is called, it is assumed that the `JMSBytesMessage` is a RECORD type message. The line "`setStructID(new String(structID, getMessageCharset()))`" sets the eye catcher.

b. The lines of code that complete the method deserialize fields in the message, in order, updating the default values set in the RECORD instance.

4. Create a put method to write the header fields to a `JMSBytesMessage`.

```
  protected BytesMessage put(MyProducer myProducer) throws IOException,
        JMSException, UnsupportedEncodingException {
     setHeaderEncoding(myProducer.getEncoding());
     setHeaderCharset(myProducer.getCharset());
     myProducer.setMQClient(true);
     BytesMessage bytes = myProducer.session.createBytesMessage();
     bytes.setStringProperty(WMQConstants.JMS_IBM_FORMAT, getHeaderFormat());
     bytes.setIntProperty(WMQConstants.JMS_IBM_ENCODING, getHeaderEncoding());
     bytes.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET,
                          myProducer.getCCSID());
     bytes.writeBytes(String.format("%1$-" + RECORD_STRUCT_ID_LENGTH + "."
                     + RECORD_STRUCT_ID_LENGTH + "s", getStructID())
                     .getBytes(getMessageCharset()), 0, RECORD_STRUCT_ID_LENGTH);
     bytes.writeInt(getVersion());
     bytes.writeInt(getStructLength());
     return bytes;
  }
```

Note:

a. `MyProducer` encapsulates the JMS `Connection`, `Destination`, `Session`, and `MessageProducer` in a single class. MyConsumer, used later on, encapsulates the JMS `Connection`, `Destination`, `Session`, and `MessageConsumer` in a single class.

b. For a `JMSBytesMessage`, if the encoding is other than `Native`, the encoding must be set in the message. The destination encoding is copied to the message encoding attribute, `JMS_IBM_CHARACTER_SET`, and saved as an attribute of the `RECORD` class.

   i) "`setMessageEncoding(myProducer.getEncoding());`" calls "`(((MQDestination) destination).getIntProperty(WMQConstants.WMQ_ENCODING));`" to get the destination encoding.

   ii) "`Bytes.setIntProperty(WMQConstants.JMS_IBM_ENCODING, getMessageEncoding());`" sets the message encoding.

c. The character set used to transform text into bytes is obtained from the destination, and saved as an attribute of the `RECORD` class. It is not set in the message, because it is not used by the IBM MQ classes for JMS when writing a `JMSBytesMessage`.

   "`messageCharset = myProducer.getCharset();`" calls

```
    public String getCharset() throws UnsupportedEncodingException,
           JMSException {
        return CCSID.getCodepage(getCCSID());
}
```

   It gets the Java character set from a coded character set identifier.

   " `CCSID.getCodepage(ccsid)` " is in the package `com.ibm.mq.headers`. The `ccsid` is obtained from another method in `MyProducer`, which queries the destination:

```
    public int getCCSID() throws JMSException {
        return (((MQDestination) destination)
                .getIntProperty(WMQConstants.WMQ_CCSID));
    }
```

d. "`myProducer.setMQClient(true);`" overrides the destination setting for the client type, forcing it to an IBM MQ MQI client. You might prefer to omit this line of code, as it obscures an administrative configuration error.

   "`myProducer.setMQClient(true);`" calls:

```
((MQDestination) destination).setTargetClient(WMQConstants.WMQ_TARGET_DEST_MQ); }
if (!getMQDest()) setMQBody();
```

The code has the side-effect of setting the IBM MQ body style to unspecified, if it must override a setting of JMS.

Note:

The IBM MQ classes for JMS write the format, encoding, and character set identifier of the message into the message descriptor, MQMD, or into the JMS header, MQRFH2. It depends on whether the message has an IBM MQ style body. Do not set the MQMD fields manually.

A method exists to set the message descriptor properties manually. It uses the JMS_IBM_MQMD_* properties. You must set the destination property, WMQ_MQMD_WRITE_ENABLED to set the JMS_IBM_MQMD_* properties:

```
((MQDestination)destination).setMQMDWriteEnabled(true);
```

You must set the destination property, WMQ_MQMD_READ_ENABLED, to read the properties.

Use the JMS_IBM_MQMD_* only if you take full control over the whole message payload. Unlike the JMS_IBM_* properties, the JMS_IBM_MQMD_* properties do not control how IBM MQ classes for JMS construct a JMS message. It is possible to create message descriptor properties that conflict with the properties of the JMS message.

e. The lines of code that completes the method serialize the attributes in class as fields in the message.

The string attributes are padded with blanks. The strings are converted to bytes using the character set defined for the record, and truncated to the length of the message fields.

5. Complete the class by adding the imports.

```
 package com.ibm.mq.id;
import java.io.IOException;
import java.io.Serializable;
import java.io.UnsupportedEncodingException;
import jakarta.jms.BytesMessage;
import jakarta.jms.JMSException;
import com.ibm.mq.constants.MQConstants;
import com.ibm.mq.headers.MQDataException;
import com.ibm.msg.client.wmq.WMQConstants;
```

6. Create a class to extend the RECORD class to include additional fields. Include a default constructor.

```
public class MyRecord extends RECORD {
   private static final long serialVersionUID = -370551723162299429L;
   private final static int FLAGS = 1;
   private final static String STRUCT_ID = "MYRD";
   private final static int DATA_LENGTH = 32;
   private final static String FORMAT = "MYRECORD";
   private int flags = FLAGS;
   private String recordData = "ABCDEFGHIJKLMNOPQRSTUVWXYZ012345";

   public MyRecord() {
       super();
       super.setStructID(STRUCT_ID);
       super.setHeaderFormat(FORMAT);
       super.setStructLength(super.getStructLength() + MQLONG_LENGTH
               + DATA_LENGTH);
   }
```

Note:

a. The RECORD subclass, MyRecord, customizes the eye catcher, format, and length of the header.

7. Create or generate the getters and setters.

a) Create the getters:

```
    public int getFlags() { return flags; }
```

```
    public String getRecordData() { return recordData; } .
```

b) Create the setters:

```
    public void setFlags(int flags) {
       this.flags = flags; }
    public void setRecordData(String recordData) {
       this.recordData = recordData; }
}
```

8. Create a constructor to create a `MyRecord` instance from a `JMSBytesMessage`.

```
public MyRecord(BytesMessage message) throws JMSException, IOException,
       MQDataException {
   super(message);
   setFlags(message.readInt());
   byte[] recordData = new byte[DATA_LENGTH];
   message.readBytes(recordData, DATA_LENGTH);
   setRecordData(new String(recordData, super.getMessageCharset()));
}
```

Note:

a. The fields that make up the standard message template are read first by the RECORD class.

b. The `recordData` text is converted to `String` using the character set property of the message.

9. Create a static method to get a message from a consumer and create a new `MyRecord` instance.

```
public static MyRecord get(MyConsumer myConsumer) throws JMSException,
       MQDataException, IOException {
   BytesMessage message = (BytesMessage) myConsumer.receive();
   return new MyRecord(message);
}
```

Note:

a. In the example, for brevity, the `MyRecord(BytesMessage)` constructor is called from the static get method. Typically, you might separate receiving the message from creating a new `MyRecord` instance.

10. Create a put method to append the customer fields to a `JMSBytesMessage` containing a message header.

```
public BytesMessage put(MyProducer myProducer) throws JMSException,
       IOException {
   BytesMessage bytes = super.put(myProducer);
   bytes.writeInt(getFlags());
   bytes.writeBytes(String.format("%1$-" + DATA_LENGTH + "."
                  + DATA_LENGTH + "s",getRecordData())
                  .getBytes(super.getMessageCharset()), 0, DATA_LENGTH);
   myProducer.send(bytes);
   return bytes;
}
```

Note:

a. The method calls in the code serialize the attributes in the MyRecord class as fields in the message.

- The `recordData` `String` attribute is padded with blanks, converted to bytes using the character set defined for the record, and truncated to the length of the `RecordData` fields.

11. Complete the class by adding the include statements.

```
 package com.ibm.mq.id;
import java.io.IOException;
import jakarta.jms.BytesMessage;
import jakarta.jms.JMSException;
import com.ibm.mq.headers.MQDataException;
```

Results

- The results from running the TryMyRecord class:
  - Sending message in coded character set 37, and using a queue manager conversion exit:

```
Out flags 1 text ABCDEFGHIJKLMNOPQRSTUVWXYZ012345 Encoding 546 CCSID 37 MQ true
Out flags 1 text ABCDEFGHIJKLMNOPQRSTUVWXYZ012345 Encoding 546 CCSID 37 MQ true
In flags 1 text ABCDEFGHIJKLMNOPQRSTUVWXYZ012345 Encoding 273 CCSID UTF-8
```

  - Sending message in coded character set 37, and not using a queue manager conversion exit:

```
Out flags 1 text ABCDEFGHIJKLMNOPQRSTUVWXYZ012345 Encoding 546 CCSID 37 MQ true
Out flags 1 text ABCDEFGHIJKLMNOPQRSTUVWXYZ012345 Encoding 546 CCSID 37 MQ true
In flags 1 text ABCDEFGHIJKLMNOPQRSTUVWXYZ012345 Encoding 546 CCSID IBM037
```

- The results from modifying the TryMyRecord class not to receive the message, and instead receiving it using the modified amqsget0.c sample. The modified sample accepts a formatted record; see Figure 38 on page 182 in "Exchanging a formatted record with a non-JMS application" on page 179.
  - Sending message in coded character set 37, and using a queue manager conversion exit:

```
Sample AMQSGET0 start
ccsid <850>, flags <1>, message <ABCDEFGHIJKLMNOPQRSTUVWXYZ012345>
no more messages
Sample AMQSGET0 end
```

  - Sending message in coded character set 37, and not using a queue manager conversion exit:

```
Sample AMQSGET0 start
MQGET ended with reason code 2110
ccsid <37>, flags <1>, message <--+-+ãÃ++ÐÊËÈiÐÎÐ+ÔÒõõµþÞÚ-±=¼¶§>
no more messages
Sample AMQSGET0 end
```

To try out the example and experiment with different code pages and a data conversion exit. Create the Java classes, configure IBM MQ, and run the main program, TryMyRecord ; see "#unique_196/unique_196_Connect_42_Try" on page 190.

1. Configure IBM MQ and JMS to run the example. The instructions are for running the example on Windows.
   a. Create a queue manager

```
crtmqm -sa -u SYSTEM.DEAD.LETTER.QUEUE QM1
strmqm QM1
```

   b. Create a queue

```
echo DEFINE QL('Q1') REPLACE | runmqsc QM1
```

   c. Create a JNDI directory

```
cd c:\
md JNDI-Directory
```

d. Switch to the JMS bin directory

   The JMS Administration program must be run from here. The path is *MQ_INSTALLATION_PATH*\java\bin.

e. Create the following JMS definitions in a file called JMSQM1Q1.txt

```
DEF CF(QM1) PROVIDERVERSION(7) QMANAGER(QM1)
DEF Q(Q1) CCSID(37) ENCODING(RRR) MSGBODY(MQ) QMANAGER(QM1) QUEUE(Q1) TARGCLIENT(MQ)
VERSION(7)
END
```

f. Run the JMSAdmin program to create the JMS resources

```
JMSAdmin < JMSQM1Q1.txt
```

2. You can create, alter, and browse the definitions you have created using IBM MQ Explorer.
3. Run TryMyRecord.

Classes used to run example

The classes listed in the following code blocks are also available in a compressed file. Download jm25529_.zip or jm25529_.tar.gz.

TryMyRecord

```
package com.ibm.mq.id;
public class TryMyRecord {
    public static void main(String[] args) throws Exception {
        MyProducer producer = new MyProducer();
        MyRecord outrec = new MyRecord();
        System.out.println("Out flags " + outrec.getFlags() + " text "
                + outrec.getRecordData() + " Encoding "
                + producer.getEncoding() + " CCSID " + producer.getCCSID()
                + " MQ " + producer.getMQDest());
        outrec.put(producer);
        System.out.println("Out flags " + outrec.getFlags() + " text "
                + outrec.getRecordData() + " Encoding "
                + producer.getEncoding() + " CCSID " + producer.getCCSID()
                + " MQ " + producer.getMQDest());
        MyRecord inrec = MyRecord.get(new MyConsumer());
        System.out.println("In  flags " + inrec.getFlags() + " text "
                + inrec.getRecordData() + " Encoding "
                + inrec.getMessageEncoding() + " CCSID "
                + inrec.getMessageCharset());
    }
}
```

RECORD

JMS 3.0

```
package com.ibm.mq.id;
import java.io.IOException;
import java.io.Serializable;
import java.io.UnsupportedEncodingException;
import jakarta.jms.BytesMessage;
import jakarta.jms.JMSException;
import com.ibm.mq.constants.MQConstants;
import com.ibm.mq.headers.MQDataException;
import com.ibm.msg.client.wmq.WMQConstants;

public abstract class RECORD implements Serializable {
    private static final long serialVersionUID = -1616617232750561712L;
    protected final static int UTF8 = 1208;
    protected final static int MQLONG_LENGTH = 4;
    protected final static int RECORD_STRUCT_ID_LENGTH = 4;
    protected final static int RECORD_VERSION_1 = 1;
    protected final String RECORD_STRUCT_ID = "BLNK";
    protected final String RECORD_TYPE = "BLANK   ";
    private String structID = RECORD_STRUCT_ID;
    private int version = RECORD_VERSION_1;
```

```
    private int structLength = RECORD_STRUCT_ID_LENGTH + MQLONG_LENGTH * 2;
    private int headerEncoding = WMQConstants.WMQ_ENCODING_NATIVE;
    private String headerCharset = "UTF-8";
    private String headerFormat = RECORD_TYPE;

    public RECORD() {
        super();
    }

    public RECORD(BytesMessage message) throws JMSException, IOException,
            MQDataException {
        super();
        setHeaderCharset(message.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET));
        setHeaderEncoding(message.getIntProperty(WMQConstants.JMS_IBM_ENCODING));
        byte[] structID = new byte[RECORD_STRUCT_ID_LENGTH];
        message.readBytes(structID, RECORD_STRUCT_ID_LENGTH);
        setStructID(new String(structID, getMessageCharset()));
        setVersion(message.readInt());
        setStructLength(message.readInt());
    }

    public String getHeaderFormat() { return headerFormat; }
    public int getHeaderEncoding() { return headerEncoding;    }
    public String getMessageCharset() { return headerCharset; }
    public int getMessageEncoding() { return headerEncoding; }
    public String getStructID() { return structID; }
    public int getStructLength() { return structLength; }
    public int getVersion() { return version; }


    protected BytesMessage put(MyProducer myProducer) throws IOException,
            JMSException, UnsupportedEncodingException {
        setHeaderEncoding(myProducer.getEncoding());
        setHeaderCharset(myProducer.getCharset());
        myProducer.setMQClient(true);
        BytesMessage bytes = myProducer.session.createBytesMessage();
        bytes.setStringProperty(WMQConstants.JMS_IBM_FORMAT, getHeaderFormat());
        bytes.setIntProperty(WMQConstants.JMS_IBM_ENCODING, getHeaderEncoding());
        bytes.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET,
                             myProducer.getCCSID());
        bytes.writeBytes(String.format("%1$-" + RECORD_STRUCT_ID_LENGTH + "."
                         + RECORD_STRUCT_ID_LENGTH + "s", getStructID())
                         .getBytes(getMessageCharset()), 0, RECORD_STRUCT_ID_LENGTH);
        bytes.writeInt(getVersion());
        bytes.writeInt(getStructLength());
        return bytes;
    }

    public void setHeaderCharset(String charset) {
        this.headerCharset = charset; }
    public void setHeaderEncoding(int encoding) {
        this.headerEncoding = encoding; }
    public void setHeaderFormat(String headerFormat) {
        this.headerFormat = headerFormat; }
    public void setStructID(String structID) {
        this.structID = structID; }
    public void setStructLength(int structLength) {
        this.structLength = structLength; }
    public void setVersion(int version) {
        this.version = version; }
}
```

JMS 2.0

```
package com.ibm.mq.id;
import java.io.IOException;
import java.io.Serializable;
import java.io.UnsupportedEncodingException;
import javax.jms.BytesMessage;
import javax.jms.JMSException;
import com.ibm.mq.constants.MQConstants;
import com.ibm.mq.headers.MQDataException;
import com.ibm.msg.client.wmq.WMQConstants;
public abstract class RECORD implements Serializable {
    private static final long serialVersionUID = -1616617232750561712L;
    protected final static int UTF8 = 1208;
    protected final static int MQLONG_LENGTH = 4;
    protected final static int RECORD_STRUCT_ID_LENGTH = 4;
    protected final static int RECORD_VERSION_1 = 1;
    protected final String RECORD_STRUCT_ID = "BLNK";
    protected final String RECORD_TYPE = "BLANK   ";
```

```
    private String structID = RECORD_STRUCT_ID;
    private int version = RECORD_VERSION_1;
    private int structLength = RECORD_STRUCT_ID_LENGTH + MQLONG_LENGTH * 2;
    private int headerEncoding = WMQConstants.WMQ_ENCODING_NATIVE;
    private String headerCharset = "UTF-8";
    private String headerFormat = RECORD_TYPE;

    public RECORD() {
        super();
    }
    public RECORD(BytesMessage message) throws JMSException, IOException,
            MQDataException {
        super();
        setHeaderCharset(message.getStringProperty(WMQConstants.JMS_IBM_CHARACTER_SET));
        setHeaderEncoding(message.getIntProperty(WMQConstants.JMS_IBM_ENCODING));
        byte[] structID = new byte[RECORD_STRUCT_ID_LENGTH];
        message.readBytes(structID, RECORD_STRUCT_ID_LENGTH);
        setStructID(new String(structID, getMessageCharset()));
        setVersion(message.readInt());
        setStructLength(message.readInt());
    }

    public String getHeaderFormat() { return headerFormat; }
    public int getHeaderEncoding() { return headerEncoding;    }
    public String getMessageCharset() { return headerCharset; }
    public int getMessageEncoding() { return headerEncoding; }
    public String getStructID() { return structID; }
    public int getStructLength() { return structLength; }
    public int getVersion() { return version; }

    protected BytesMessage put(MyProducer myProducer) throws IOException,
            JMSException, UnsupportedEncodingException {
        setHeaderEncoding(myProducer.getEncoding());
        setHeaderCharset(myProducer.getCharset());
        myProducer.setMQClient(true);
        BytesMessage bytes = myProducer.session.createBytesMessage();
        bytes.setStringProperty(WMQConstants.JMS_IBM_FORMAT, getHeaderFormat());
        bytes.setIntProperty(WMQConstants.JMS_IBM_ENCODING, getHeaderEncoding());
        bytes.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET,
                             myProducer.getCCSID());
        bytes.writeBytes(String.format("%1$-" + RECORD_STRUCT_ID_LENGTH + "."
                         + RECORD_STRUCT_ID_LENGTH + "s", getStructID())
                         .getBytes(getMessageCharset()), 0, RECORD_STRUCT_ID_LENGTH);
        bytes.writeInt(getVersion());
        bytes.writeInt(getStructLength());
        return bytes;
    }

    public void setHeaderCharset(String charset) {
        this.headerCharset = charset; }
    public void setHeaderEncoding(int encoding) {
        this.headerEncoding = encoding; }
    public void setHeaderFormat(String headerFormat) {
        this.headerFormat = headerFormat; }
    public void setStructID(String structID) {
        this.structID = structID; }
    public void setStructLength(int structLength) {
        this.structLength = structLength; }
    public void setVersion(int version) {
        this.version = version; }
}
```

MyRecord

JMS 3.0

```
package com.ibm.mq.id;
import java.io.IOException;
import jakarta.jms.BytesMessage;
import jakarta.jms.JMSException;
import com.ibm.mq.headers.MQDataException;

public class MyRecord extends RECORD {
    private static final long serialVersionUID = -370551723162299429L;
    private final static int FLAGS = 1;
    private final static String STRUCT_ID = "MYRD";
    private final static int DATA_LENGTH = 32;
    private final static String FORMAT = "MYRECORD";
    private int flags = FLAGS;
    private String recordData = "ABCDEFGHIJKLMNOPQRSTUVWXYZ012345";
```

```
    public MyRecord() {
        super();
        super.setStructID(STRUCT_ID);
        super.setHeaderFormat(FORMAT);
        super.setStructLength(super.getStructLength() + MQLONG_LENGTH
                + DATA_LENGTH);
    }

    public MyRecord(BytesMessage message) throws JMSException, IOException,
            MQDataException {
        super(message);
        setFlags(message.readInt());
        byte[] recordData = new byte[DATA_LENGTH];
        message.readBytes(recordData, DATA_LENGTH);
        setRecordData(new String(recordData, super.getMessageCharset()));
    }

    public static MyRecord get(MyConsumer myConsumer) throws JMSException,
            MQDataException, IOException {
        BytesMessage message = (BytesMessage) myConsumer.receive();
        return new MyRecord(message);
    }

    public int getFlags() { return flags; }
    public String getRecordData() { return recordData; } .


    public BytesMessage put(MyProducer myProducer) throws JMSException,
            IOException {
        BytesMessage bytes = super.put(myProducer);
        bytes.writeInt(getFlags());
        bytes.writeBytes(String.format("%1$-" + DATA_LENGTH + "."
                        + DATA_LENGTH + "s",getRecordData())
                        .getBytes(super.getMessageCharset()), 0, DATA_LENGTH);
        myProducer.send(bytes);
        return bytes;
    }

    public void setFlags(int flags) {
        this.flags = flags; }
    public void setRecordData(String recordData) {
        this.recordData = recordData; }
}
```

JMS 2.0

```
package com.ibm.mq.id;
import java.io.IOException;
import javax.jms.BytesMessage;
import javax.jms.JMSException;
import com.ibm.mq.headers.MQDataException;
public class MyRecord extends RECORD {
    private static final long serialVersionUID = -370551723162299429L;
    private final static int FLAGS = 1;
    private final static String STRUCT_ID = "MYRD";
    private final static int DATA_LENGTH = 32;
    private final static String FORMAT = "MYRECORD";
    private int flags = FLAGS;
    private String recordData = "ABCDEFGHIJKLMNOPQRSTUVWXYZ012345";

    public MyRecord() {
        super();
        super.setStructID(STRUCT_ID);
        super.setHeaderFormat(FORMAT);
        super.setStructLength(super.getStructLength() + MQLONG_LENGTH
                + DATA_LENGTH);
    }
    public MyRecord(BytesMessage message) throws JMSException, IOException,
            MQDataException {
        super(message);
        setFlags(message.readInt());
        byte[] recordData = new byte[DATA_LENGTH];
        message.readBytes(recordData, DATA_LENGTH);
        setRecordData(new String(recordData, super.getMessageCharset()));
    }
    public static MyRecord get(MyConsumer myConsumer) throws JMSException,
            MQDataException, IOException {
        BytesMessage message = (BytesMessage) myConsumer.receive();
        return new MyRecord(message);
    }
    public int getFlags() { return flags; }
```

```
    public String getRecordData() { return recordData; } .

    public BytesMessage put(MyProducer myProducer) throws JMSException,
            IOException {
        BytesMessage bytes = super.put(myProducer);
        bytes.writeInt(getFlags());
        bytes.writeBytes(String.format("%1$-" + DATA_LENGTH + "."
                        + DATA_LENGTH + "s",getRecordData())
                        .getBytes(super.getMessageCharset()), 0, DATA_LENGTH);
        myProducer.send(bytes);
        return bytes;
    }
    public void setFlags(int flags) {
        this.flags = flags; }
    public void setRecordData(String recordData) {
        this.recordData = recordData; }
}
```

EndPoint

JMS 3.0

```
package com.ibm.mq.id;
import java.io.UnsupportedEncodingException;
import jakarta.jms.Connection;
import jakarta.jms.ConnectionFactory;
import jakarta.jms.Destination;
import jakarta.jms.JMSException;
import jakarta.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.ibm.mq.headers.CCSID;
import com.ibm.mq.jms.MQDestination;
import com.ibm.msg.client.wmq.WMQConstants;
public abstract class EndPoint {
    public Context ctx;
    public ConnectionFactory cf;
    public Connection connection;
    public Destination destination;
    public Session session;
    protected EndPoint() throws NamingException, JMSException {
        System.setProperty("java.naming.provider.url", "file:/C:/JNDI-Directory");
        System.setProperty("java.naming.factory.initial",
                           "com.sun.jndi.fscontext.RefFSContextFactory");
        ctx = new InitialContext();
        cf = (ConnectionFactory) ctx.lookup("QM1");
        connection = cf.createConnection();
        destination = (Destination) ctx.lookup("Q1");
        ((MQDestination)destination).setReceiveConversion
                                    (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE); }
    protected EndPoint(String cFactory, String dest) throws NamingException,
            JMSException {
        System.setProperty("java.naming.provider.url", "file:/C:/JNDI-Directory");
        System.setProperty("java.naming.factory.initial",
                           "com.sun.jndi.fscontext.RefFSContextFactory");
        ctx = new InitialContext();
        cf = (ConnectionFactory) ctx.lookup(cFactory);
        connection = cf.createConnection();
        destination = (Destination) ctx.lookup(dest);
        ((MQDestination)destination).setReceiveConversion
                                    (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);  }
    public int getCCSID() throws JMSException {
        return (((MQDestination) destination)
                .getIntProperty(WMQConstants.WMQ_CCSID));  }
    public String getCharset() throws UnsupportedEncodingException,
            JMSException {
        return CCSID.getCodepage(getCCSID());  }
    public int getEncoding() throws JMSException {
        return (((MQDestination) destination)
                .getIntProperty(WMQConstants.WMQ_ENCODING));  }
    public boolean getMQDest() throws JMSException {
        if ((((MQDestination) destination).getMessageBodyStyle()
               == WMQConstants.WMQ_MESSAGE_BODY_MQ)
            || ((((MQDestination) destination).getMessageBodyStyle()
                   == WMQConstants.WMQ_MESSAGE_BODY_UNSPECIFIED)
               && (((MQDestination) destination).getTargetClient()
                   == WMQConstants.WMQ_TARGET_DEST_MQ)))
            return true;
```

```
        else
            return false; }
    public void setCCSID(int ccsid) throws JMSException {
        ((MQDestination) destination).setIntProperty(WMQConstants.WMQ_CCSID,
                ccsid); }
    public void setEncoding(int encoding) throws JMSException {
        ((MQDestination) destination).setIntProperty(WMQConstants.WMQ_ENCODING,
                encoding); }
    public void setMQBody() throws JMSException {
        ((MQDestination) destination)
                    .setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_UNSPECIFIED);  }
    public void setMQBody(boolean mqbody) throws JMSException {
        if (mqbody)  ((MQDestination) destination)
                    .setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_MQ);
        else         ((MQDestination) destination)
                    .setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_JMS);  }
    public void setMQClient(boolean mqclient) throws JMSException {
       if (mqclient){
            ((MQDestination) destination).setTargetClient(WMQConstants.WMQ_TARGET_DEST_MQ);
            if (!getMQDest()) setMQBody();
        }
       else
            ((MQDestination) destination).setTargetClient(WMQConstants.WMQ_TARGET_DEST_JMS); }
    }
```

JMS 2.0

```
package com.ibm.mq.id;
import java.io.UnsupportedEncodingException;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.ibm.mq.headers.CCSID;
import com.ibm.mq.jms.MQDestination;
import com.ibm.msg.client.wmq.WMQConstants;
public abstract class EndPoint {
    public Context ctx;
    public ConnectionFactory cf;
    public Connection connection;
    public Destination destination;
    public Session session;
    protected EndPoint() throws NamingException, JMSException {
        System.setProperty("java.naming.provider.url", "file:/C:/JNDI-Directory");
        System.setProperty("java.naming.factory.initial",
                           "com.sun.jndi.fscontext.RefFSContextFactory");
        ctx = new InitialContext();
        cf = (ConnectionFactory) ctx.lookup("QM1");
        connection = cf.createConnection();
        destination = (Destination) ctx.lookup("Q1");
        ((MQDestination)destination).setReceiveConversion
                                    (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE); }
    protected EndPoint(String cFactory, String dest) throws NamingException,
            JMSException {
        System.setProperty("java.naming.provider.url", "file:/C:/JNDI-Directory");
        System.setProperty("java.naming.factory.initial",
                           "com.sun.jndi.fscontext.RefFSContextFactory");
        ctx = new InitialContext();
        cf = (ConnectionFactory) ctx.lookup(cFactory);
        connection = cf.createConnection();
        destination = (Destination) ctx.lookup(dest);
        ((MQDestination)destination).setReceiveConversion
                                    (WMQConstants.WMQ_RECEIVE_CONVERSION_QMGR);
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);  }
    public int getCCSID() throws JMSException {
        return (((MQDestination) destination)
                .getIntProperty(WMQConstants.WMQ_CCSID));  }
    public String getCharset() throws UnsupportedEncodingException,
            JMSException {
        return CCSID.getCodepage(getCCSID());  }
    public int getEncoding() throws JMSException {
        return (((MQDestination) destination)
                .getIntProperty(WMQConstants.WMQ_ENCODING));  }
    public boolean getMQDest() throws JMSException {
        if ((((MQDestination) destination).getMessageBodyStyle()
               == WMQConstants.WMQ_MESSAGE_BODY_MQ)
```

```
            || ((((MQDestination) destination).getMessageBodyStyle()
                   == WMQConstants.WMQ_MESSAGE_BODY_UNSPECIFIED)
               && (((MQDestination) destination).getTargetClient()
                   == WMQConstants.WMQ_TARGET_DEST_MQ)))
            return true;
        else
            return false; }
    public void setCCSID(int ccsid) throws JMSException {
        ((MQDestination) destination).setIntProperty(WMQConstants.WMQ_CCSID,
                ccsid); }
    public void setEncoding(int encoding) throws JMSException {
        ((MQDestination) destination).setIntProperty(WMQConstants.WMQ_ENCODING,
                encoding); }
    public void setMQBody() throws JMSException {
        ((MQDestination) destination)
                    .setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_UNSPECIFIED);  }
    public void setMQBody(boolean mqbody) throws JMSException {
        if (mqbody)  ((MQDestination) destination)
                    .setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_MQ);
        else         ((MQDestination) destination)
                    .setMessageBodyStyle(WMQConstants.WMQ_MESSAGE_BODY_JMS);  }
    public void setMQClient(boolean mqclient) throws JMSException {
       if (mqclient){
            ((MQDestination) destination).setTargetClient(WMQConstants.WMQ_TARGET_DEST_MQ);
            if (!getMQDest()) setMQBody();
        }
       else
            ((MQDestination) destination).setTargetClient(WMQConstants.WMQ_TARGET_DEST_JMS); }
    }
```

MyProducer

JMS 3.0

```
package com.ibm.mq.id;
import jakarta.jms.JMSException;
import jakarta.jms.Message;
import jakarta.jms.MessageProducer;
import javax.naming.NamingException;
public class MyProducer extends EndPoint {
    public MessageProducer producer;
    public MyProducer() throws NamingException, JMSException {
        super();
        producer = session.createProducer(destination); }
    public MyProducer(String cFactory, String dest) throws NamingException,
            JMSException {
        super(cFactory, dest);
        producer = session.createProducer(destination); }
    public void send(Message message) throws JMSException {
        producer.send(message); }
}
```

JMS 2.0

```
package com.ibm.mq.id;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.naming.NamingException;
public class MyProducer extends EndPoint {
    public MessageProducer producer;
    public MyProducer() throws NamingException, JMSException {
        super();
        producer = session.createProducer(destination); }
    public MyProducer(String cFactory, String dest) throws NamingException,
            JMSException {
        super(cFactory, dest);
        producer = session.createProducer(destination); }
    public void send(Message message) throws JMSException {
        producer.send(message); }
}
```

MyConsumer

JMS 3.0

```
package com.ibm.mq.id;
import jakarta.jms.JMSException;
import jakarta.jms.Message;
```

```
import jakarta.jms.MessageConsumer;
import javax.naming.NamingException;
public class MyConsumer extends EndPoint {
    public MessageConsumer consumer;
    public MyConsumer() throws NamingException, JMSException {
        super();
        consumer = session.createConsumer(destination);
        connection.start();  }
    public MyConsumer(String cFactory, String dest) throws NamingException,
            JMSException {
        super(cFactory, dest);
        consumer = session.createConsumer(destination);
        connection.start();  }
    public Message receive() throws JMSException {
        return consumer.receive();  }
}
```

JMS 2.0

```
package com.ibm.mq.id;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;
public class MyConsumer extends EndPoint {
    public MessageConsumer consumer;
    public MyConsumer() throws NamingException, JMSException {
        super();
        consumer = session.createConsumer(destination);
        connection.start();  }
    public MyConsumer(String cFactory, String dest) throws NamingException,
            JMSException {
        super(cFactory, dest);
        consumer = session.createConsumer(destination);
        connection.start();  }
    public Message receive() throws JMSException {
        return consumer.receive();  }
}
```

Creating and configuring connection factories and destinations

An IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging application can create connection factories and destinations by retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace, by using the IBM JMS extensions, or by using the IBM MQ JMS extensions. An application can also use the IBM JMS extensions or IBM MQ JMS extensions to set the properties of connection factories and destinations.

Connection factories and destinations are starting points in the flow of logic of a JMS or Jakarta Messaging application. An application uses a ConnectionFactory object to create a connection to a messaging server, and uses a Queue or Topic object as a target to send messages to or a source from which to receive messages. An application therefore needs to create at least one connection factory and one or more destinations. Having created a connection factory or destination, the application might then need to configure the object by setting one or more of its properties.

In summary, an application can create and configure connection factories and destinations in the following ways:

Using JNDI to retrieve administered objects

An administrator can use the IBM MQ JMS administration tool as described in Configuring JMS and Jakarta Messaging objects using the administration tools, or IBM MQ Explorer as described in Configuring JMS 2.0 objects using IBM MQ Explorer, to create and configure connection factories and destinations as administered objects in a JNDI namespace. An application can then retrieve the administered objects from the JNDI namespace. Having retrieved an administered object, the application can, if required, set or change one or more of its properties by using either the IBM JMS extensions or the IBM MQ JMS extensions.

Note: JMS 3.0 For Jakarta Messaging 3.0, you cannot administer JNDI using IBM MQ Explorer. JNDI administration is supported by the Jakarta Messaging 3.0 variant of **JMSAdmin**, which is **JMS30Admin**.

Using the IBM JMS extensions

An application can use the IBM JMS extensions to create connection factories and destinations dynamically at run time. The application first creates a JmsFactoryFactory object, and then uses methods of this object to create connection factories and destinations. Having created a connection factory or destination, the application can use methods inherited from the JmsPropertyContext interface to set its properties. Alternatively, the application can use a uniform resource identifier (URI) to specify one or more properties of a destination when it creates the destination.

Using the IBM MQ JMS extensions

An application can also use the IBM MQ JMS extensions to create connection factories and destinations dynamically at run time. The application uses the supplied constructors to create connection factories and destinations. Having created a connection factory or destination, the application can use methods of the object to set its properties. Alternatively, the application can use a URI to specify one or more properties of a destination when it creates the destination.

Related tasks

Configuring JMS and Jakarta Messaging resources

Using JNDI to retrieve administered objects in a JMS or Jakarta Messaging application

To retrieve administered objects from a Java Naming and Directory Interface (JNDI) namespace, a JMS or Jakarta Messaging application must create an initial context and then use the lookup() method to retrieve the objects.

Before an application can retrieve administered objects from a JNDI namespace, an administrator must first create the administered objects.

JMS 2.0 For JMS 2.0, the administrator can use the IBM MQ JMS administration tool, **JMSAmin**, or IBM MQ Explorer to create and maintain administered objects in a JNDI namespace. For more information, see Configuring connection factories and destinations in a JNDI namespace.

JMS 3.0 For Jakarta Messaging 3.0, you cannot administer JNDI using IBM MQ Explorer. JNDI administration is supported by the Jakarta Messaging 3.0 variant of **JMSAdmin**, which is **JMS30Admin**.

An application server, typically provides its own repository for administered objects and its own tools for creating and maintaining the objects.

To retrieve administered objects from a JNDI namespace, an application must first create an initial context, as shown in the following example:

JMS 3.0

```
import jakarta.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
.
.
.
String url = "ldap://server.company.com/o=company_us,c=us";
String icf = "com.sun.jndi.ldap.LdapCtxFactory";
.
java.util.Hashtable environment = new java.util.Hashtable();
environment.put(Context.PROVIDER_URL, url);
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
Context ctx = new InitialDirContext(environment);
```

JMS 2.0

```
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
.
.
.
String url = "ldap://server.company.com/o=company_us,c=us";
String icf = "com.sun.jndi.ldap.LdapCtxFactory";
.
java.util.Hashtable environment = new java.util.Hashtable();
environment.put(Context.PROVIDER_URL, url);
```

```
environment.put(Context.INITIAL_CONTEXT_FACTORY, icf);
Context ctx = new InitialDirContext(environment);
```

In this code, the String variables `url` and `icf` have the following meanings:

url
The uniform resource locator (URL) of the directory service. The URL can have one of the following formats:

- `ldap://`*`hostname`*`/`*`contextName`* , for a directory service based on an LDAP server
- `file:/`*`directoryPath`* , for a directory service based on the local file system

icf
The class name of the initial context factory, which can be one of the following values:

- `com.sun.jndi.ldap.LdapCtxFactory`, for a directory service based on an LDAP server
- `com.sun.jndi.fscontext.RefFSContextFactory`, for a directory service based on the local file system

Note that some combinations of a JNDI package and a Lightweight Directory Access Protocol (LDAP) service provider can cause LDAP error 84 to occur. To resolve this problem, insert the following line of code before the call to InitialDirContext():

```
environment.put(Context.REFERRAL, "throw");
```

After an initial context is obtained, the application can retrieve administered objects from the JNDI namespace by using the lookup() method, as shown in the following example:

```
ConnectionFactory factory;
Queue queue;
Topic topic;
.
.
.
factory = (ConnectionFactory)ctx.lookup("cn=myCF");
queue = (Queue)ctx.lookup("cn=myQ");
topic = (Topic)ctx.lookup("cn=myT");
```

This code retrieves the following objects from an LDAP based namespace:

- A ConnectionFactory object bound with the name myCF
- A Queue object bound with the name myQ
- A Topic object bound with the name myT

For more information about using JNDI, see the JNDI documentation provided by Oracle Corporation.

Related tasks
Configuring JMS 2.0 objects using IBM MQ Explorer
Configuring JMS and Jakarta Messaging objects using the administration tools
Configuring JMS 2.0 resources in WebSphere Application Server

Using the IBM JMS extensions
IBM MQ classes for JMS (JMS 2.0) and IBM MQ classes for Jakarta Messaging (Jakarta Messaging 3.0) each contain a functionally-identical set of extensions to the JMS API called the IBM JMS extensions. An application can use these extensions to create connection factories and destinations dynamically at run time, and to set the properties of IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging objects. The extensions can be used with any messaging provider.

The IBM JMS extensions are a set of interfaces and classes in the following packages:

- com.ibm.msg.client.jms
- com.ibm.msg.client.services

For Jakarta Messaging 3.0, these packages are in `com.ibm.jakarta.client.jar`.

JMS 2.0 For JMS 2.0, these packages are in `com.ibm.mqjms.jar` or `com.ibm.mq.allclient.jar`.

These extensions provide the following function:

- A factory-based mechanism for creating connection factories and destinations dynamically at run time, instead of retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace
- A set of methods for setting the properties of IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging objects
- A set of exception classes with methods for obtaining detailed information about a problem
- A set of methods for controlling tracing
- A set of methods for obtaining version information about IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging

For creating connection factories and destinations dynamically at run time, and setting and getting their properties, the IBM JMS extensions provide an alternative set of interfaces to the IBM MQ JMS extensions. However, whereas the IBM MQ JMS extensions are specific to the IBM MQ messaging provider, the IBM JMS extensions are not specific to IBM MQ and can be used with any messaging provider within the layered architecture described in IBM MQ classes for JMS architecture.

The interface com.ibm.msg.client.wmq.WMQConstants (JMS 2.0) or com.ibm.msg.jakarta.client.wmq.WMQConstants (Jakarta Messaging 3.0) contains the definitions of constants that an application can use when setting the properties of IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging objects using the IBM JMS extensions. The interface contains constants for the IBM MQ messaging provider and JMS constants that are independent of any messaging provider.

The examples of code that follow assume that the following import statements are included in the Java class:

JMS 3.0
```
import com.ibm.msg.jakarta.client.jms.*;
import com.ibm.msg.jakarta.client.services.*;
import com.ibm.msg.jakarta.client.wmq.WMQConstants;
```

JMS 2.0
```
import com.ibm.msg.client.jms.*;
import com.ibm.msg.client.services.*;
import com.ibm.msg.client.wmq.WMQConstants;
```

Creating connection factories and destinations

Before an application can create connection factories and destinations using the IBM JMS extensions, it must first create a JmsFactoryFactory object. To create a JmsFactoryFactory object, the application calls the getInstance() method of the JmsFactoryFactory class, as shown in the following example:

JMS 3.0
```
JmsFactoryFactory ff = JmsFactoryFactory.getInstance(JmsConstants.JAKARTA_WMQ_PROVIDER);
```

JMS 2.0
```
JmsFactoryFactory ff = JmsFactoryFactory.getInstance(JmsConstants.WMQ_PROVIDER);
```

The parameter on the getInstance() call is a constant that identifies the IBM MQ messaging provider as the chosen messaging provider. The application can then use the JmsFactoryFactory object to create connection factories and destinations.

To create a connection factory, the application calls the createConnectionFactory() method of the JmsFactoryFactory object, as shown in the following example:

```
JmsConnectionFactory factory = ff.createConnectionFactory();
```

This statement creates a JmsConnectionFactory object with the default values for all its properties, which means that the application connects to the default queue manager in bindings mode. If you want an application to connect in client mode, or connect to a queue manager other than the default queue manager, the application must set the appropriate properties of the JmsConnectionFactory object before creating the connection. For information about how to do this, see "Setting the properties of IBM MQ classes for JMS objects" on page 202.

The JmsFactoryFactory class also contains methods to create connection factories of the following types:

- JmsQueueConnectionFactory
- JmsTopicConnectionFactory
- JmsXAConnectionFactory
- JmsXAQueueConnectionFactory
- JmsXATopicConnectionFactory

To create a Queue object, the application calls the createQueue() method of the JmsFactoryFactory object, as shown in the following example:

```
JmsQueue q1 = ff.createQueue("Q1");
```

This statement creates an JmsQueue object with the default values for all its properties. The object represents an IBM MQ queue called Q1 that belongs to the local queue manager. This queue can be a local queue, an alias queue, or a remote queue definition.

The createQueue() method can also accept a queue uniform resource identifier (URI) as a parameter. A queue URI is a string that specifies the name of an IBM MQ queue and, optionally, the name of the queue manager that owns the queue, and one or more properties of the JmsQueue object. The following statement contains an example of a queue URI:

```
JmsQueue q2 = ff.createQueue("queue://QM2/Q2?persistence=2&priority=5");
```

The JmsQueue object created by this statement represents an IBM MQ queue called Q2 that is owned by queue manager QM2, and all messages sent to this destination are persistent and have a priority of 5. For more information about queue URIs, see "Uniform resource identifiers (URIs)" on page 214. For an alternative way of setting the properties of a JmsQueue object, see "Setting the properties of IBM MQ classes for JMS objects" on page 202.

To create a Topic object, an application can use the createTopic() method of the JmsFactoryFactory object, as shown in the following example:

```
JmsTopic t1 = ff.createTopic("Sport/Football/Results");
```

This statement creates a JmsTopic object with the default values for all its properties. The object represents a topic called Sport/Football/Results.

The createTopic() method can also accept a topic URI as a parameter. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the JmsTopic object. The following statements contain an example of a topic URI:

```
String s1 = "topic://Sport/Tennis/Results?persistence=1&priority=0";
JmsTopic t2 = ff.createTopic(s1);
```

The JmsTopic object created by these statements represents a topic called Sport/Tennis/Results, and all messages sent to this destination are nonpersistent and have a priority of 0. For more information about topic URIs, see "Uniform resource identifiers (URIs)" on page 214. For an alternative way of setting the properties of a JmsTopic object, see "Setting the properties of IBM MQ classes for JMS objects" on page 202.

After an application has created a connection factory or destination, that object can be used only with the selected messaging provider.

Setting the properties of IBM MQ classes for JMS objects

To set the properties of IBM MQ classes for JMS objects using the IBM JMS extensions, an application uses the methods of the com.ibm.msg.client.JmsPropertyContext interface. Similarly, to set the properties of IBM MQ classes for Jakarta Messaging objects using the IBM JMS extensions, an application uses the methods of the com.ibm.msg.jakarta.client.JmsPropertyContext interface.

For each Java data type, the JmsPropertyContext interface contains a method to set the value of a property with that data type, and a method to get the value of a property with that data type. For example, an application calls the setIntProperty() method to set a property with an integer value, and calls the getIntProperty() method to get a property with an integer value.

Instances of classes in the com.ibm.mq.jms and com.ibm.mq.jakarta.jms packages inherit the methods of the corresponding JmsPropertyContext interfaces. An application can therefore use these methods to set the properties of MQConnectionFactory, MQQueue, and MQTopic objects.

When an application creates an IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging object, any properties with default values are set automatically. When an application sets a property, the new value replaces any previous value the property had. After a property has been set, it cannot be deleted, but its value can be changed.

If an application attempts to set a property to a value that is not a valid value for the property, IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging throws a JMSException exception. If an application attempts to get a property that has not been set, the behavior is as described in the JMS specification. IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging throw a NumberFormatException exception for primitive data types and return null for referenced data types.

In addition to the predefined properties of an IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging object, an application can set its own properties. These application-defined properties are ignored by IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

For more information about the properties of IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging objects, see Properties of IBM MQ classes for JMS objects.

The following code is an example of how to set properties using the IBM JMS extensions. The code sets five properties of a connection factory.

```
factory.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE,
WMQConstants.WMQ_CM_CLIENT);
factory.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "QM1");
factory.setStringProperty(WMQConstants.WMQ_HOST_NAME, "HOST1");
factory.setIntProperty(WMQConstants.WMQ_PORT, 1415);
factory.setStringProperty(WMQConstants.WMQ_CHANNEL, "QM1.SVR");
factory.setStringProperty(WMQConstants.WMQ_APPLICATIONNAME, "My Application");
```

The effect of setting these properties is that the application connects to queue manager QM1 in client mode, using an MQI channel called QM1.SVR. The queue manager is running on a system with host name HOST1, and the listener for the queue manager is listening in port number 1415. This connection and other queue manager connections associated with sessions under it, have the application name "My Application" associated with them.

Note: Queue managers running on z/OS platforms do not support setting application names, and this setting is therefore ignored.

The JmsPropertyContext interface also contains the setObjectProperty() method, which an application can use to set properties. The second parameter of the method is an object that encapsulates the value of the property. For example, the following code creates an Integer object that encapsulates the integer 1415, and then calls setObjectProperty() to set the PORT property of a connection factory to the value 1415:

```
Integer port = new Integer(1415);
factory.setObjectProperty(WMQConstants.WMQ_PORT, port);
```

This code is therefore equivalent to the following statement:

```
factory.setIntProperty(WMQConstants.WMQ_PORT, 1415);
```

Conversely, the getObjectProperty() method returns an object that encapsulates the value of a property.

Implicit conversion of a property value from one data type to another

When an application uses a method of the JmsPropertyContext interface to set or get the property of an IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging object, the value of the property can be implicitly converted from one data type to another.

For example, the following statement sets the PRIORITY property of the JmsQueue object q1:

```
q1.setStringProperty(WMQConstants.WMQ_PRIORITY, "5");
```

The PRIORITY property has an integer value, and so the setStringProperty() call implicitly converts the string "5" (the source value) to the integer 5 (the target value), which then becomes the value of the PRIORITY property.

Conversely, the following statement gets the PRIORITY property of the JmsQueue object q1:

```
String s1 = q1.getStringProperty(WMQConstants.WMQ_PRIORITY);
```

The integer 5 (the source value), which is the value of the PRIORITY property, is implicitly converted to the string "5" (the target value) by the getStringProperty() call.

The conversions supported by IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are shown in Table 34 on page 203.

Table 34. Supported conversions from one data type to another

Source data type	**Supported target data types**
boolean	String
byte	int, long, short, String
char	String
double	String
float	double, String
int	long, String
long	String
short	int, long, String
String	boolean, byte, double, float, int, long, short

The general rules governing the supported conversions are as follows:

- Numeric values can be converted from one data type to another provided no data is lost during the conversion. For example, a value with data type `int` can be converted into a value with data type `long`, but cannot be converted into a value with data type `short`.
- A value of any data type can be converted into a string.
- A string can be converted to a value of any other data type (except `char`) provided the string is in the correct format for the conversion. If an application attempts to convert a string that is not in the correct format, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging throw a NumberFormatException exception.
- If an application attempts a conversion that is not supported, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging throw a MessageFormatException exception.

The specific rules for converting a value from one data type to another are as follows:

- When converting a boolean value to a string, the value `true` is converted to the string "true", and the value `false` is converted to the string "false".
- When converting a string to a boolean value, the string "true" (not case-sensitive) is converted to `true`, and the string "false" (not case-sensitive) is converted to `false`. Any other string is converted to `false`.
- When converting a string to a value with data type `byte`, `int`, `long`, or `short`, the string must have the following format:

  [*blanks*][*sign*] *digits*

  The meanings of the components of the string are as follows:

  blanks
  Optional leading blank characters.

  sign
  An optional plus sign (+) or minus sign (-).

  digits
  A contiguous sequence of digits (0-9). At least one digit must be present.

  After the sequence of digits, the string can contain other characters that are not digits, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal integer.

  If the string is not in the correct format, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging throw a NumberFormatException exception.
- When converting a string to a value with data type `double` or `float`, the string must have the following format:

  [*blanks*][*sign*] *digits* [*e_char* [*e_sign*] *e_digits*]

  The meanings of the components of the string are as follows:

  blanks
  Optional leading blank characters.

  sign
  An optional plus sign (+) or minus sign (-).

  digits
  A contiguous sequence of digits (0-9). At least one digit must be present.

  e_char
  An exponent character, which is either *E* or *e*.

  e_sign
  An optional plus sign (+) or minus sign (-) for the exponent.

  e_digits
  A contiguous sequence of digits (0-9) for the exponent. At least one digit must be present if the string contains an exponent character.

After the sequence of digits, or the optional characters representing an exponent, the string can contain other characters that are not digits, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal floating point number with an exponent that is a power of 10.

If the string is not in the correct format, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging throw a NumberFormatException exception.

- When converting a numeric value (including a value with data type `byte`) to a string, the value is converted to the string representation of the value as a decimal number, not the string containing the ASCII character for that value. For example, the integer 65 is converted to the string "65", not the string "A".

Setting more than one property in a single call

The JmsPropertyContext interface also contains the setBatchProperties() method, which an application can use to set more than one property in a single call. The parameter of the method is a Map object that encapsulates a set of property name-value pairs.

For example, the following code uses the setBatchProperties() method to set the same five properties of a connection factory as shown in “Setting the properties of IBM MQ classes for JMS objects” on page 202. The code creates an instance of the HashMap class, which implements the Map interface.

```
HashMap batchProperties = new HashMap();
batchProperties.put(WMQConstants.WMQ_CONNECTION_MODE,
                    new Integer(WMQConstants.WMQ_CM_CLIENT));
batchProperties.put(WMQConstants.WMQ_QUEUE_MANAGER, "QM1");
batchProperties.put(WMQConstants.WMQ_WMQ_HOST_NAME, "HOST1");
batchProperties.put(WMQConstants.WMQ_PORT, "1414");
batchProperties.put(WMQConstants.WMQ_CHANNEL, "QM1.SVR");
factory.setBatchProperties(batchProperties);
```

Note that the second parameter of the Map.put() method must be an object. Therefore a property value with a primitive data type must be encapsulated within an object or represented by a string, as shown in the example.

The setBatchProperties() method validates each property. If the setBatchProperties() method cannot set a property because, for example, its value is not valid, none of the specified properties are set.

Property names and values

If an application uses the methods of the appropriate JmsPropertyContext interface to set and get the properties of IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging objects, the application can specify the names and values of properties in any of the following ways. Each of the accompanying examples shows how to set the PRIORITY property of the JmsQueue object q1 so that a message sent to the queue has the priority specified on the send() call.

Using the property names and values that are defined as constants in the com.ibm.msg.client.wmq.WMQConstants interface

The following statement is an example of how to specify the names and values of properties in this way:

```
q1.setIntProperty(WMQConstants.WMQ_PRIORITY, WMQConstants.WMQ_PRI_APP);
```

Using the property names and values that can be used in queue and topic uniform resource identifiers (URIs)

The following statement is an example of how to specify the names and values of properties in this way:

```
q1.setIntProperty("priority", -2);
```

Only the names and values of properties of destinations can be specified in this way.

Using the property names and values that are recognized by the IBM MQ JMS administration tool

The following statement is an example of how to specify the names and values of properties in this way:

```
q1.setStringProperty("PRIORITY", "APP");
```

The short form of the property name is also acceptable, as shown in the following statement:

```
q1.setStringProperty("PRI", "APP");
```

When an application gets a property, the value returned depends on the way in which the application specifies the name of the property. For example, if an application specifies the constant WMQConstants.WMQ_PRIORITY as the property name, the value returned is the integer -2:

```
int n1 = getIntProperty(WMQConstants.WMQ_PRIORITY);
```

The same value is returned if the application specifies the string "priority" as the property name:

```
int n2 = getIntProperty("priority");
```

However, if the application specifies the string "PRIORITY" or "PRI" as the property name, the value returned is the string "APP":

```
String s1 = getStringProperty("PRI");
```

Internally, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging store property names and values as the literal values defined in the matching WMQConstants interface. This is the defined canonical format for property names and values. As a general rule, if an application sets properties using one of the other two ways of specifying property names and values, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging have to convert the names and values from the specified input format into the canonical format. Similarly, if an application gets properties using one of the other two ways of specifying property names and values, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging must convert the names from the specified input format into the canonical format, and convert the values from the canonical format into the required output format. Having to perform these conversions might have implications for performance.

Property names and values returned by exceptions, in trace files, or in the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging log, are always in the canonical format.

Using the Map interface

The JmsPropertyContext interface extends the java.util.Map interface. An application can therefore use the methods of the Map interface to access the properties of an IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging object.

For example, the following code prints out the names and values of all the properties of a connection factory. The code uses only the methods of the Map interface to get the names and values of the properties.

```
// Get the names of all the properties
Set propNames = factory.keySet();

// Loop round all the property names and get the property values
Iterator iterator = propNames.iterator();
while (iterator.hasNext()){
    String pName = (String)iterator.next();
    System.out.println(pName+"="+factory.get(pName));
}
```

Using the methods of the Map interface does not bypass any property validations or conversions.

Using the IBM MQ JMS extensions
IBM MQ classes for JMS contains a set of extensions to the JMS API called the IBM MQ JMS extensions. An application can use these extensions to create connection factories and destinations dynamically at run time, and to set the properties of connection factories and destinations.

IBM MQ classes for JMS contains a set of classes in the packages com.ibm.jms and com.ibm.mq.jms. These classes implement the JMS interfaces and contain the IBM MQ JMS extensions. The examples of code that follow assume that these packages have been imported by the following statements:

```
import com.ibm.jms.*;
import com.ibm.mq.jms.*;
import com.ibm.msg.client.wmq.WMQConstants;
```

An application can use the IBM MQ JMS extensions to perform the following functions:

- Create connection factories and destinations dynamically at run time, instead of retrieving them as administered objects from a Java Naming and Directory Interface (JNDI) namespace
- Set the properties of connection factories and destinations

Creating connection factories

To create a connection factory, an application can use the MQConnectionFactory constructor, as shown in the following example:

```
MQConnectionFactory factory = new MQConnectionFactory();
```

This statement creates an MQConnectionFactory object with the default values for all its properties, which means that the application connects to the default queue manager in bindings mode. If you want an application to connect in client mode, or connect to a queue manager other than the default queue manager, the application must set the appropriate properties of the MQConnectionFactory object before creating the connection. For information about how to do this, see “Setting the properties of connection factories” on page 207.

An application can create connection factories of the following types in a similar way:

- MQQueueConnectionFactory
- MQTopicConnectionFactory
- MQXAConnectionFactory
- MQXAQueueConnectionFactory
- MQXATopicConnectionFactory

Setting the properties of connection factories

An application can set the properties of a connection factory by calling the appropriate methods of the connection factory. The connection factory can either be an administered object or an object created dynamically at run time.

Consider the following code, for example:

```
MQConnectionFactory factory = new MQConnectionFactory();
.
factory.setTransportType(WMQConstants.WMQ_CM_CLIENT);
factory.setQueueManager("QM1");
factory.setHostName("HOST1");
factory.setPort(1415);
factory.setChannel("QM1.SVR");
```

This code creates an MQConnectionFactory object and then sets five properties of the object. The effect of setting these properties is that the application connects to queue manager QM1 in client mode using an MQI channel called QM1.SVR. The queue manager is running on a system with host name HOST1, and the listener for the queue manager is listening in port number 1415.

An application that uses a real-time connection to a broker can use only the publish/subscribe style of messaging. It cannot use the point-to-point style of messaging.

Only certain combinations of properties of a connection factory are valid. For information about which combinations are valid, see Dependencies between properties of IBM MQ classes for JMS objects.

For more information about the properties of a connection factory, and the methods used to set its properties, see Properties of IBM MQ classes for JMS objects.

Creating destinations

To create a Queue object, an application can use the MQQueue constructor, as shown in the following example:

```
MQQueue q1 = new MQQueue("Q1");
```

This statement creates an MQQueue object with the default values for all its properties. The object represents an IBM MQ queue called Q1 that belongs to the local queue manager. This queue can be a local queue, an alias queue, or a remote queue definition.

An alternative form of the MQQueue constructor has two parameters, as shown in the following example:

```
MQQueue q2 = new MQQueue("QM2", "Q2");
```

The MQQueue object created by this statement represents an IBM MQ queue called Q2 that is owned by queue manager QM2. The queue manager identified in this way can be the local queue manager or a remote queue manager. If it is a remote queue manager, IBM MQ must be configured so that, when the application sends a message to this destination, WebSphere MQ can route the message from the local queue manager to the remote queue manager.

The MQQueue constructor can also accept a queue uniform resource identifier (URI) as a single parameter. A queue URI is a string that specifies the name of an IBM MQ queue and, optionally, the name of the queue manager that owns the queue, and one or more properties of the MQQueue object. The following statement contains an example of a queue URI:

```
MQQueue q3 = new MQQueue("queue://QM3/Q3?persistence=2&priority=5");
```

The MQQueue object created by this statement represents an IBM MQ queue called Q3 that is owned by queue manager QM3, and all messages sent to this destination are persistent and have a priority of 5. For more information about queue URIs, see “Uniform resource identifiers (URIs)” on page 214. For an alternative way of setting the properties of an MQQueue object, see “Setting the properties of destinations” on page 209.

To create a Topic object, an application can use the MQTopic constructor, as shown in the following example:

```
MQTopic t1 = new MQTopic("Sport/Football/Results");
```

This statement creates an MQTopic object with the default values for all its properties. The object represents a topic called Sport/Football/Results.

The MQTopic constructor can also accept a topic URI as a parameter. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the MQTopic object. The following statement contains an example of a topic URI:

```
MQTopic t2 = new MQTopic("topic://Sport/Tennis/Results?persistence=1&priority=0");
```

The MQTopic object created by this statement represents a topic called Sport/Tennis/Results, and all messages sent to this destination are nonpersistent and have a priority of 0. For more information about topic URIs, see “Uniform resource identifiers (URIs)” on page 214. For an alternative way of setting the properties of an MQTopic object, see “Setting the properties of destinations” on page 209.

Setting the properties of destinations

An application can set the properties of a destination by calling the appropriate methods of the destination. The destination can either be an administered object or an object created dynamically at run time.

Consider the following code, for example:

```
MQQueue q1 = new MQQueue("Q1");
.
q1.setPersistence(WMQConstants.WMQ_PER_PER);
q1.setPriority(5);
```

This code creates an MQQueue object and then sets two properties of the object. The effect of setting these properties is that all messages sent to the destination are persistent and have a priority of 5.

An application can set the properties of MQTopic object in a similar way, as shown in the following example:

```
MQTopic t1 = new MQTopic("Sport/Football/Results");
.
t1.setPersistence(WMQConstants.WMQ_PER_NON);
t1.setPriority(0);
```

This code creates an MQTopic object and then sets two properties of the object. The effect of setting these properties is that all messages sent to the destination are nonpersistent and have a priority of 0.

For more information about the properties of a destination, and the methods used to set its properties, see Properties of IBM MQ classes for JMS objects.

Linux AIX *Connecting to IBM MQ from a JMS application*

To build a connection, a JMS application uses a **ConnectionFactory** object to create a **Connection** object, then starts the connection.

For JMS 2.0 and later, applications usually connect to a messaging provider using a **ConnectionFactory** object and the `createContext()` method.

In earlier versions of JMS, you had to first use `createConnection` to create a **Connection** object, then start the connection call `getSession()` to create a **Session** object that could carry out messaging operations.

A **JMSContext** object effectively encapsulates both the **Connection** and **Session** objects. If you want to use the traditional approach and create the connection and session objects directly, see “Building a connection in a JMS application” on page 210 and “Creating a session in a JMS application” on page 211.

To create a **JMSContext** object, an application uses the `createContext()` method of a **ConnectionFactory** object, as shown in the following example:

```
ConnectionFactory factory;
Connection connection;
.
.
.
connection = factory.createContext();
```

When a JMS connection is created, the IBM MQ classes for JMS create a connection handle (Hconn) and starts a conversation with the queue manager.

Note: Note that the application process ID is used as the default user identity to be passed to the queue manager. If the application is running in client transport mode then this process ID must exist, with the relevant authorizations, on the server. If you want a different identity to be used, then use the `createConnection(username,password)` method.

V 9.4.0 This mechanism can also be used to supply an authentication token, see Obtaining an authentication token from your chosen token issuer.

JMS 1.0 *Building a connection in a JMS application*

To build a connection in JMS 1.0, a JMS application uses a ConnectionFactory object to create a Connection object and then starts the connection.

To create a Connection object, an application uses the createConnection() method of a ConnectionFactory object, as shown in the following example:

```
ConnectionFactory factory;
Connection connection;
.
.
.
connection = factory.createConnection();
```

When a JMS connection is created, the IBM MQ classes for JMS creates a connection handle (Hconn) and starts a conversation with the queue manager.

The QueueConnectionFactory interface and the TopicConnectionFactory interface each inherits the createConnection() method from the ConnectionFactory interface. You can therefore use the createConnection() method to create a domain specific object, as shown in the following example:

```
QueueConnectionFactory qcf;
Connection connection;
.
.
.
connection = qcf.createConnection();
```

This fragment of code creates a QueueConnection object. An application can now perform a domain independent operation on this object, or an operation that is applicable only to the point-to-point domain. However, if the application attempts to perform an operation that is applicable only to the publish/subscribe domain, an IllegalStateException exception is thrown with the following message:

```
JMSMQ1112: Operation for a domain specific object was not valid.
           Operation createProducer() is not valid for type com.ibm.mq.jms.MQTopic
```

This is because the connection was created from a domain specific connection factory.

Note: Note that the application process ID is used as the default user identity to be passed to the queue manager. If the application is running in client transport mode then this process ID must exist, with the relevant authorizations, on the server. If you want a different identity to be used, then use the createConnection(username, password) method.

The JMS specification states that a connection is created in the `stopped` state. Until a connection starts, a message consumer that is associated with the connection cannot receive any messages. To start a connection, an application uses the start() method of a Connection object, as shown in the following example:

```
connection.start();
```

V 9.4.0 This mechanism can also be used to supply an authentication token, see Obtaining an authentication token from your chosen token issuer.

JMS 1.0 *Creating a session in a JMS application*

To create a session in JMS 1.0, a JMS application uses the createSession() method of a Connection object.

The createSession() method has two parameters:

1. A parameter that specifies whether the session is transacted or not transacted
2. A parameter that specifies the acknowledgment mode for the session

For example, the following code creates a session that is not transacted and has an acknowledgment mode of AUTO_ACKNOWLEDGE:

```
Session session;
.
boolean transacted = false;
session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
```

When a JMS session is created, the IBM MQ classes for JMS creates a connection handle (Hconn) and starts a conversation with the queue manager.

A Session object, and any MessageProducer or MessageConsumer object created from it, cannot be used concurrently by different threads of a multithreaded application. The simplest way of ensuring that these objects are not used concurrently is to create a separate Session object for each thread.

V 9.4.0 This mechanism can also be used to supply an authentication token, see Obtaining an authentication token from your chosen token issuer.

Transacted sessions in JMS applications

JMS applications can run local transactions by first creating a transacted session. An application can commit or roll back a transaction.

JMS applications can run local transactions. A local transaction is a transaction that involves changes only to the resources of the queue manager to which the application is connected. To run local transactions, an application must first create a transacted session by calling the createSession() method of a Connection object, specifying as a parameter that the session is transacted. Subsequently, all messages sent and received within the session are grouped into a sequence of transactions. A transaction ends when the application commits or rolls back the messages it has sent and received since the transaction began.

To commit a transaction, an application calls the commit() method of the Session object. When a transaction is committed, all messages sent within the transaction become available for delivery to other applications, and all messages received within the transaction are acknowledged so that the messaging server does not attempt to deliver them to the application again. In the point-to-point domain, the messaging server also removes the received messages from their queues.

To roll back a transaction, an application calls the rollback() method of the Session object. When a transaction is rolled back, all messages sent within the transaction are discarded by the messaging server, and all messages received within the transaction become available for delivery again. In the point-to-point domain, the messages that were received are put back on their queues and become visible to other applications again.

A new transaction starts automatically when an application creates a transacted session or calls the commit() or rollback() method. Therefore, a transacted session always has an active transaction.

When an application closes a transacted session, an implicit rollback occurs. When an application closes a connection, an implicit rollback occurs for all the connection's transacted sessions.

If an application ends without closing a connection, an implicit rollback also occurs for all the connection's transacted sessions.

A transaction is wholly contained within a transacted session. A transaction cannot span sessions. This means that it is not possible for an application to send and receive messages in two or more transacted sessions and then commit or roll back all these actions as a single transaction.

Acknowledgment modes of JMS sessions
Every session that is not transacted has an acknowledgment mode that determines how messages received by the application are acknowledged. Three acknowledgment modes are available, and the choice of acknowledgment mode affects the design of the application.

If a session is not transacted, the way that messages received by the application are acknowledged is determined by the acknowledgment mode of the session. The three acknowledgment modes are described in the following paragraphs:

AUTO_ACKNOWLEDGE

The session automatically acknowledges each message received by the application.

If messages are delivered synchronously to the application, the session acknowledges receipt of a message every time a Receive call completes successfully. If messages are delivered asynchronously, the session acknowledges receipt of a message every time a call to the onMessage() method of a message listener completes successfully.

If the application receives a message successfully, but a failure prevents acknowledgment from occurring, the message becomes available for delivery again. The application must therefore be able to handle a message that is re-delivered.

DUPS_OK_ACKNOWLEDGE

The session acknowledges the messages received by the application at times it selects.

Using this acknowledgment mode reduces the amount of work the session must do, but a failure that prevents message acknowledgment might result in more than one message becoming available for delivery again. The application must therefore be able to handle messages that are re-delivered.

Restriction: In AUTO_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE modes, JMS does not support an application throwing an unhandled exception in a message listener. This means that messages are always acknowledged when the message listener returns, regardless of whether it was processed successfully (provided any failures are non-fatal and do not prevent the application from continuing). If you require finer control of message acknowledgment, use the CLIENT_ACKNOWLEDGE or transacted modes, which give the application full control of the acknowledgment functions.

CLIENT_ACKNOWLEDGE

The application acknowledges the messages it receives by calling the Acknowledge method of the Message class.

The application can acknowledge the receipt of each message individually, or it can receive a batch of messages and call the Acknowledge method only for the last message it receives. When the Acknowledge method is called all messages received since the last time the method was called are acknowledged.

In conjunction with any of these acknowledgment modes, an application can stop and restart the delivery of messages in a session by calling the Recover method of the Session class. Messages received but previously unacknowledged are re-delivered. However, they might not be delivered in the same sequence in which they were previously delivered. In the meantime, higher priority messages might have arrived, and some of the original messages might have expired. In the point-to-point domain, some of the original messages might have been consumed by another application.

An application can determine whether a message is being re-delivered by examining the contents of the JMSRedelivered header field of the message. The application does this by calling the getJMSRedelivered() method of the Message class.

Creating destinations in a JMS application

Instead of retrieving destinations as administered objects from a Java Naming and Directory Interface (JNDI) namespace, a JMS application can use a session to create destinations dynamically at run time.

An application can use a uniform resource identifier (URI) to identify an IBM MQ queue or a topic and, optionally, to specify one or more properties of a Queue or Topic object.

Using a session to create Queue objects

To create a Queue object, an application can use the createQueue() method of a Session object, as shown in the following example:

```
Session session;
.
Queue q1 = session.createQueue("Q1");
```

This code creates a Queue object with the default values for all its properties. The object represents an IBM MQ queue called Q1 that belongs to the local queue manager. This queue can be a local queue, an alias queue, or a remote queue definition.

The createQueue() method also accepts a queue URI as a parameter. A queue URI is a string that specifies the name of an IBM MQ queue and, optionally, the name of the queue manager that owns the queue and one or more properties of the Queue object. The following statement contains an example of a queue URI:

```
Queue q2 = session.createQueue("queue://QM2/Q2?persistence=2&priority=5");
```

The Queue object created by this statement represents an IBM MQ queue called Q2 that is owned by a queue manager called QM2, and all messages sent to this destination are persistent and have a priority of 5. The queue manager identified in this way can be the local queue manager or a remote queue manager. If it is a remote queue manager, IBM MQ must be configured so that, when the application sends a message to this destination, WebSphere MQ can route the message from the local queue manager to queue manager QM2. For more information about URIs, see “Uniform resource identifiers (URIs)” on page 214.

Note that the parameter on the createQueue() method contains provider specific information. Therefore, using the createQueue() method to create a Queue object, instead of retrieving a Queue object as an administered object from a JNDI namespace, might make your application less portable.

An application can create a TemporaryQueue object by using the createTemporaryQueue() method of a Session object, as shown in the following example:

```
TemporaryQueue q3 = session.createTemporaryQueue();
```

Although a session is used to create a temporary queue, the scope of a temporary queue is the connection that was used to create the session. Any of the connection's sessions can create message producers and message consumers for the temporary queue. The temporary queue remains until the connection ends or the application explicitly deletes the temporary queue by using the TemporaryQueue.delete() method, whichever is the sooner.

When an application creates a temporary queue, IBM MQ classes for JMS creates a dynamic queue in the queue manager to which the application is connected. The TEMPMODEL property of the connection factory specifies the name of the model queue that is used to create the dynamic queue, and the TEMPQPREFIX property of the connection factory specifies the prefix that is used to form the name of the dynamic queue.

Using a session to create Topic objects

To create a Topic object, an application can use the createTopic() method of a Session object, as shown in the following example:

```
Session session;
```

```
.
Topic t1 = session.createTopic("Sport/Football/Results");
```

This code creates an Topic object with the default values for all its properties. The object represents a topic called Sport/Football/Results.

The createTopic() method also accepts a topic URI as a parameter. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the Topic object. The following code contains an example of a topic URI:

```
String uri = "topic://Sport/Tennis/Results?persistence=1&priority=0";
Topic t2 = session.createTopic(uri);
```

The Topic object created by this code represents a topic called Sport/Tennis/Results, and all messages sent to this destination are nonpersistent and have a priority of 0. For more information about topic URIs, see “Uniform resource identifiers (URIs)” on page 214.

Note that the parameter on the createTopic() method contains provider specific information. Therefore, using the createTopic() method to create a Topic object, instead of retrieving a Topic object as an administered object from a JNDI namespace, might make your application less portable.

An application can create a TemporaryTopic object by using the createTemporaryTopic() method of a Session object, as shown in the following example:

```
TemporaryTopic t3 = session.createTemporaryTopic();
```

Although a session is used to create a temporary topic, the scope of a temporary topic is the connection that was used to create the session. Any of the connection's sessions can create message producers and message consumers for the temporary topic. The temporary topic remains until the connection ends or the application explicitly deletes the temporary topic by using the TemporaryTopic.delete() method, whichever is the sooner.

When an application creates a temporary topic, IBM MQ classes for JMS creates a topic with a name that commences with the characters TEMP/*tempTopicPrefix*, where *tempTopicPrefix* is the value of the TEMPTOPICPREFIX property of the connection factory.

Uniform resource identifiers (URIs)

A queue URI is a string that specifies the name of an IBM MQ queue and, optionally, the name of the queue manager that owns the queue and one or more properties of the Queue object created by the application. A topic URI is a string that specifies the name of a topic and, optionally, one or more properties of the Topic object created by the application.

A queue URI has the following format:

```
queue://[ qMgrName ]/qName [? propertyName1 = propertyValue1
& propertyName2 = propertyValue2
&...]
```

A topic URI has the following format:

```
topic://topicName [? propertyName1 = propertyValue1
& propertyName2 = propertyValue2
&...]
```

The variables in these formats have the following meanings:

qMgrName
The name of the queue manager that owns the queue identified by the URI.

The queue manager can the local queue manager or a remote queue manager. If it is a remote queue manager, IBM MQ must be configured so that, when an application sends a message to the queue, WebSphere MQ can route the message from the local queue manager to the remote queue manager.

If no name is specified, the local queue manager is assumed.

qName
The name of the IBM MQ queue.

The queue can be a local queue, an alias queue, or a remote queue definition.

For the rules for creating queue names, see Rules for naming IBM MQ objects.

topicName
The name of the topic.

For the rules for creating topic names, see Rules for naming IBM MQ objects. Avoid the use of the wildcard characters +, #, *, and ? in topic names. Topic names containing these characters can cause unexpected results when you subscribe to them. See Combining topic strings.

***propertyName1*, *propertyName2*, ...**
The names of the properties of the Queue or Topic object created by the application. Table 35 on page 215 lists the valid property names that can be used in a URI.

If no properties are specified, the Queue or Topic object has the default values for all its properties.

***propertyValue1*, *propertyValue2*, ...**
The values of the properties of the Queue or Topic object created by the application. Table 35 on page 215 lists the valid property values that can be used in a URI.

Brackets ([]) denotes an optional component, and the ellipsis (...) means that the list of property name-value pairs, if present, can contain one or more name-value pairs.

Table 35 on page 215 lists the valid property names and valid values that can be used in queue and topic URIs. Although the IBM MQ JMS administration tool uses symbolic constants for the values of properties, URIs cannot contain symbolic constants.

Table 35. Property names and valid values for use in queue and topic URIs

Property name	Description	Valid values
CCSID	How the character data in the body of a message is represented when IBM MQ classes for JMS forwards the message to the destination	• Any coded character set identifier supported by IBM MQ.
encoding	How the numeric data in the body of a message is represented when IBM MQ classes for JMS forwards the message to the destination	• Any valid value for the *Encoding* field in an IBM MQ message descriptor.
expiry	The time to live for messages sent to the destination	• -2 - As specified on the send() call or, if not specified on the send() call, the default time to live of the message producer. • 0 - A message sent to the destination never expires. • A positive integer specifying the time to live in milliseconds.

Table 35. Property names and valid values for use in queue and topic URIs (continued)

Property name	Description	Valid values
multicast	The multicast setting for a topic when using a real-time connection to a broker	The following list contains the valid values. Associated with each value is the corresponding value of the MULTICAST property as used in the IBM MQ JMS administration tool. For a description of the MULTICAST property and its valid values, see Properties of IBM MQ classes for JMS objects. • -1 - ASCF • 0 - DISABLED • 3 - NOTR • 5 - RELIABLE • 7 - ENABLED
persistence	The persistence of messages sent to the destination	• -2 - As specified on the send() call or, if not specified on the send() call, the default persistence of the message producer. • -1 - As specified by the *DefPersistence* attribute of the IBM MQ queue or topic. • 1 - Nonpersistent. • 2 - Persistent. • 3 - Equivalent to the value HIGH for the PERSISTENCE property as used in the IBM MQ JMS administration tool. For an explanation of this value, see "JMS persistent messages" on page 245.
priority	The priority of messages sent to the destination	• -2 - As specified on the send() call or, if not specified on the send() call, the default priority of the message producer. • -1 - As specified by the *DefPriority* attribute of the IBM MQ queue or topic. • An integer in the range 0-9 specifying the priority of messages sent to the destination.
targetClient	Whether messages sent to the destination contain an MQRFH2 header	• 0 - Messages contain an MQRFH2 header. • 1 - Messages do not contain an MQRFH2 header.

For example, the following URI identifies an IBM MQ queue called Q1 that is owned by the local queue manager. A Queue object created using this URI has the default values for all its properties.

```
queue:///Q1
```

The following URI identifies an IBM MQ queue called Q2 that is owned by a queue manager called QM2. All messages sent to this destination have a priority of 6. The remaining properties of the Queue object created using this URI have their default values.

```
queue://QM2/Q2?priority=6
```

The following URI identifies a topic called Sport/Athletics/Results. All messages sent to this destination are nonpersistent and have a priority of 0. The remaining properties of the Topic object created using this URI have their default values.

```
topic://Sport/Athletics/Results?persistence=1&priority=0
```

Sending messages in a JMS application

Before a JMS application can send messages to a destination, it must first create a MessageProducer object for the destination. To send a message to the destination, the application creates a Message object and then calls the send() method of the MessageProducer object.

An application uses a MessageProducer object to send messages. An application normally creates a MessageProducer object for a specific destination, which can be a queue or a topic, so that all messages sent using the message producer are sent to the same destination. Therefore, before an application can create a MessageProducer object, it must first create a Queue or Topic object. For information about how to create a Queue or Topic object, see the following topics:

- "Using JNDI to retrieve administered objects in a JMS or Jakarta Messaging application" on page 198
- "Using the IBM JMS extensions" on page 199
- "Using the IBM MQ JMS extensions" on page 207
- "Creating destinations in a JMS application" on page 212

To create a MessageProducer object, an application uses the createProducer() method of a Session object, as shown in the following example:

```
MessageProducer producer = session.createProducer(destination);
```

The parameter `destination` is a Queue or Topic object that the application has created previously.

Before an application can send a message, it must create a Message object. The body of a message contains the application data, and JMS defines five types of message body:

- Bytes
- Map
- Object
- Stream
- Text

Each type of message body has its own JMS interface, which is a sub-interface of the Message interface, and a method in the Session interface for creating a message with that type of body. For example, the interface for a text message is called TextMessage, and an application uses the createTextMessage() method of a Session object to create a text message, as shown in the following statement:

```
TextMessage outMessage = session.createTextMessage(outString);
```

For more information about messages and message bodies, see “JMS messages” on page 138.

To send a message, an application uses the send() method of a MessageProducer object, as shown in the following example:

```
producer.send(outMessage);
```

An application can use the send() method to send messages in either messaging domain. The nature of the destination determines which messaging domain is used. However, TopicPublisher, the sub-interface of MessageProducer that is specific to the publish/subscribe domain, also has a publish() method, which can be used instead of the send() method. The two methods are functionally the same.

An application can create a MessageProducer object with no specified destination. In this case, the application must specify the destination when calling the send() method.

If an application sends a message within a transaction, the message is not delivered to its destination until the transaction is committed. This means that an application cannot send a message and receive a reply to the message within the same transaction.

A destination can be configured so that when an application sends messages to it, IBM MQ classes for JMS forwards the message and returns control back to the application without determining whether the queue manager has received the message safely. This is sometimes referred to as *asynchronous put*. For more information, see “Putting messages asynchronously in IBM MQ classes for JMS” on page 308.

Receiving messages in a JMS application

An application uses a message consumer to receive messages. A durable topic subscriber is a message consumer that receives all messages sent to a destination, including those sent while the consumer is inactive. An application can select which messages it wants to receive by using a message selector, and can receive messages asynchronously by using a message listener.

An application uses a MessageConsumer object to receive messages. An application creates a MessageConsumer object for a specific destination, which can be a queue or a topic, so that all messages received using the message consumer are received from the same destination. Therefore, before an application can create a MessageConsumer object, it must first create a Queue or Topic object. For information about how to create a Queue or Topic object, see the following topics:

- “Using JNDI to retrieve administered objects in a JMS or Jakarta Messaging application” on page 198
- “Using the IBM JMS extensions” on page 199
- “Using the IBM MQ JMS extensions” on page 207
- “Creating destinations in a JMS application” on page 212

To create a MessageConsumer object, an application uses the createConsumer() method of a Session object, as shown in the following example:

```
MessageConsumer consumer = session.createConsumer(destination);
```

The parameter `destination` is a Queue or Topic object that the application has created previously.

The application then uses the receive() method of the MessageConsumer object to receive a message from the destination, as shown in the following example:

```
Message inMessage = consumer.receive(1000);
```

The parameter on the receive() call specifies how long in milliseconds the method waits for a suitable message to arrive if no message is available immediately. If you omit this parameter, the call blocks indefinitely until a suitable message arrives. If you do not want the application to wait for a message, use the receiveNoWait() method instead.

The receive() method returns a message of a specific type. For example, when an application receives a text message, the object returned by the receive() call is a TextMessage object.

However, the declared type of object returned by a receive() call is a Message object. Therefore, in order to extract the data from the body of a message that has just been received, the application must cast from the Message class to the more specific subclass, such as TextMessage. If the type of the message is not known, the application can use the `instanceof` operator to determine the type. It is always good practice for an application to determine the type of a message before casting so that errors can be handled gracefully.

The following code uses the `instanceof` operator and shows how to extract the data from the body of a text message:

```
if (inMessage instanceof TextMessage) {
  String replyString = ((TextMessage) inMessage).getText();
  .
  .
  .
} else {
  // Print error message if Message was not a TextMessage.
  System.out.println("Reply message was not a TextMessage");
}
```

If an application sends a message within a transaction, the message is not delivered to its destination until the transaction is committed. This means that an application cannot send a message and receive a reply to the message within the same transaction.

If a message consumer receives messages from a destination that is configured for read ahead, any nonpersistent messages that are in the read ahead buffer when the application ends are discarded.

In the publish/subscribe domain, JMS identifies two types of message consumer, nondurable topic subscriber and durable topic subscriber, which are described in the following two sections.

Nondurable topic subscribers

A nondurable topic subscriber receives only those messages that are published while the subscriber is active. A nondurable subscription starts when an application creates a nondurable topic subscriber and ends when the application closes the subscriber, or when the subscriber falls out of scope. As an extension in IBM MQ classes for JMS, a nondurable topic subscriber also receives retained publications.

To create a nondurable topic subscriber, an application can use the domain independent createConsumer() method, specifying a Topic object as the destination. Alternatively, an application can use the domain specific createSubscriber() method, as shown in the following example:

```
TopicSubscriber subscriber = session.createSubscriber(topic);
```

The parameter `topic` is a Topic object that the application has created previously.

Durable topic subscribers

Restriction: An application cannot create durable topic subscribers when using a real-time connection to a broker.

A durable topic subscriber receives all messages that are published during the life of a durable subscription. These messages include all those that are published while the subscriber is not active. As an extension in IBM MQ classes for JMS, a durable topic subscriber also receives retained publications.

To create a durable topic subscriber, an application uses the createDurableSubscriber() method of a Session object, as shown in the following example:

```
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "D_SUB_000001");
```

On the createDurableSubscriber() call, the first parameter is a Topic object that the application has created previously, and the second parameter is a name that is used to identify the durable subscription.

The session used to create a durable topic subscriber must have an associated client identifier. The client identifier associated with a session is the same as the client identifier for the connection that is used to create the session. The client identifier can be specified by setting the CLIENTID property of the ConnectionFactory object. Alternatively, an application can specify the client identifier by calling the setClientID() method of the Connection object.

The name that is used to identify a durable subscription must be unique only within the client identifier, and therefore the client identifier forms part of the full, unique identifier of a durable subscription. To continue using a durable subscription that was created previously, an application must create a durable topic subscriber using a session with the same client identifier as that associated with the durable subscription, and using the same subscription name.

A durable subscription starts when an application creates a durable topic subscriber using a client identifier and subscription name for which no durable subscription currently exists. However, a durable subscription does not end when the application closes the durable topic subscriber. To end a durable subscription, an application must call the unsubscribe() method of a Session object that has the same client identifier as that associated with the durable subscription. The parameter on the unsubscribe() call is the subscription name, as shown in the following example:

```
session.unsubscribe("D_SUB_000001");
```

The scope of a durable subscription is a queue manager. If a durable subscription exists on one queue manager, and an application connected to another queue manager creates a durable subscription with the same client identifier and subscription name, the two durable subscriptions are completely independent.

Message selectors

An application can specify that only those messages that satisfy certain criteria are returned by successive receive() calls. When creating a MessageConsumer object, the application can specify a Structured Query Language (SQL) expression that determines which messages are retrieved. This SQL expression is called a *message selector*. The message selector can contain the names of JMS message header fields and message properties. For information about how to construct a message selector, see “Message selectors in JMS” on page 139.

The following example shows how an application can select messages based on a user defined property called myProp:

```
MessageConsumer consumer;
.
consumer = session.createConsumer(destination, "myProp = 'blue'");
```

The JMS specification does not allow an application to change the message selector of a message consumer. After an application creates a message consumer with a message selector, the message selector remains for the life of that consumer. If an application requires more than one message selector, the application must create a message consumer for each message selector.

Note that, when an application is connected to a Version 7 queue manager, the MSGSELECTION property of the connection factory has no effect. To optimize performance, all message selection is done by the queue manager.

Suppressing local publications

An application can create a message consumer that ignores publications published on the consumer's own connection. The application does this by setting the third parameter on a createConsumer() call to `true`, as shown in the following example:

```
MessageConsumer consumer = session.createConsumer(topic, null, true);
```

On a createDurableSubscriber() call, the application does this by setting the fourth parameter to `true`, as shown in the following example

```
String selector = "company = 'IBM'";
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "D_SUB_000001",
                                                              selector, true);
```

Asynchronous delivery of messages

An application can receive messages asynchronously by registering a message listener with a message consumer. The message listener has a method called onMessage, which is called asynchronously when a suitable message is available and whose purpose is to process the message. The following code illustrates the mechanism:

JMS 3.0

```
import jakarta.jms.*;

public class MyClass implements MessageListener
{
  // The method that is called asynchronously when a suitable message is available
  public void onMessage(Message message)
  {
    System.out.println("Message is "+message);

    // The code to process the message
    .
    .
    .
  }
}
.
.
.
// Main program (possibly in another class)
.
// Creating the message listener
MyClass listener = new MyClass();

// Registering the message listener with a message consumer
consumer.setMessageListener(listener);

// The main program now continues with other processing
```

JMS 2.0

```
import javax.jms.*;

public class MyClass implements MessageListener
{
  // The method that is called asynchronously when a suitable message is available
  public void onMessage(Message message)
  {
    System.out.println("Message is "+message);

    // The code to process the message
    .
    .
    .
  }
}
.
.
.
// Main program (possibly in another class)
.
// Creating the message listener
MyClass listener = new MyClass();

// Registering the message listener with a message consumer
consumer.setMessageListener(listener);

// The main program now continues with other processing
```

An application can use a session either for receiving messages synchronously using receive() calls, or for receiving messages asynchronously using message listeners, but not for both. If an application needs to receive messages synchronously and asynchronously, it must create separate sessions.

Once a session is set up to receive messages asynchronously, the following methods cannot be called on that session or on objects created from that session:

- MessageConsumer.receive()
- MessageConsumer.receive(long)
- MessageConsumer.receiveNoWait()
- Session.acknowledge()
- MessageProducer.send(Destination, Message)
- MessageProducer.send(Destination, Message, int, int, long)
- MessageProducer.send(Message)
- MessageProducer.send(Message, int, int, long)
- MessageProducer.send(Destination, Message, CompletionListener)
- MessageProducer.send(Destination, Message, int, int, long, CompletionListener)
- MessageProducer.send(Message, CompletionListener)
- MessageProducer.send(Message, int, int, long, CompletionListener)
- Session.commit()
- Session.createBrowser(Queue)
- Session.createBrowser(Queue, String)
- Session.createBytesMessage()
- Session.createConsumer(Destination)
- Session.createConsumer(Destination, String, boolean)
- Session.createDurableSubscriber(Topic, String)
- Session.createDurableSubscriber(Topic, String, String, boolean)
- Session.createMapMessage()
- Session.createMessage()
- Session.createObjectMessage()
- Session.createObjectMessage(Serializable)
- Session.createProducer(Destination)
- Session.createQueue(String)
- Session.createStreamMessage()
- Session.createTemporaryQueue()
- Session.createTemporaryTopic()
- Session.createTextMessage()
- Session.createTextMessage(String)
- Session.createTopic()
- Session.getAcknowledgeMode()
- Session.getMessageListener()
- Session.getTransacted()
- Session.rollback()
- Session.unsubscribe(String)

If any of these methods are called, a JMSException containing the message:

```
JMSCC0033: A synchronous method call is not permitted when a session is being used
asynchronously:'method name'
```

is thrown.

Receiving poison messages

An application can receive a message that cannot be processed. There can be several reasons why the message cannot be processed, for example the message might have an incorrect format. Such messages are described as poison messages and require special handling to prevent the message being recursively processed.

For details on how to handle poison messages, see "Handling poison messages in IBM MQ classes for JMS" on page 225.

Tailoring buffer sizes to suit the messages being received

When a message is received from IBM MQ by a non-JMS application a message buffer must be provided by the application for the message to be written into. JMS applications do not need to manually create a buffer. The IBM MQ classes for JMS automatically create and size message buffers to suit the sizes of the messages being received. For most applications, automatically managed buffers provide a suitable balance of performance and convenience for the application developer. In certain circumstances, it might be beneficial to specify the initial size of the message buffer manually. The default initial size of an IBM MQ JMS receive buffer is 4 KB. If an application is always going to receive messages that are 256 KB in size, it might be preferable to configure the initial buffer size to 256 KB. This can avoid the need for the IBM MQ classes for JMS to attempt and fail to receive the message into a 4 KB buffer before resizing it to 256 KB and successfully receiving it. For a client-connected application, this can avoid the need for a potentially wasted network round trip while the IBM MQ classes for JMS determine the correct buffer size to use.

The initial buffer size can be configured by setting the com.ibm.mq.jmqi.defaultMaxMsgSize Java property to your chosen value, in bytes. Note that this property affects all IBM MQ JMS applications that are running inside the Java Virtual Machine, so be careful not to adversely affect other message consumers that receive messages of a different size.

The IBM MQ classes for JMS still attempt to automatically reduce the size of the buffer if several messages smaller than the configured size are received. By default, this happens if 10 messages are received which are all smaller than the buffer size. For example, if 10 messages are received in a row that are 128 KB in size, the buffer is reduced to from 256 KB to 128 KB. It is then increased again when larger messages are received. It is possible to configure the number of messages that must be received before a buffer is reduced in size. For example, this might be useful if the application is known to receive five large messages followed by 10 smaller messages and then another five large messages. With the default settings, the buffer would be reduced after the 10 smaller messages had been received and would need increasing again for the larger messages. The Java system property com.ibm.mq.jmqi.smallMsgBufferReductionThreshold can be set to the number of messages that must be received before the size of the buffer is reduced. In this example, it could be set to 20 to prevent 10 smaller messages from reducing the buffer size.

The properties can be set independently of each other. For example, you might choose to leave the initial buffer size to its default value of 4 KB but increase the value of com.ibm.mq.jmqi.smallMsgBufferReductionThreshold so once the buffer is increased in size it stays that size for longer.

If large numbers of MQRC_TRUNCATED_MSG_FAILED (2080) return codes are seen for your JMS applications in MQI statistics records, this might be an indication that you would benefit from configuring a higher initial buffer size for those applications, or reducing the frequency with which buffer sizes are reduced. However, it is important to note that for a long running application you are likely to see only a very small number of MQRC_TRUNCATED_MSG_FAILED return codes. This is because typically the buffer is increased to the correct size immediately after the first large message is received, and is not reduced in size unless a number of smaller messages are received. It is therefore possible that a large number of

MQRC_TRUNCATED_MSG_FAILED indicates other poor application practices such as connecting to IBM MQ to receive just one or two messages before disconnecting.

Retrieval of subscription user data

If the messages that an IBM MQ classes for JMS application is consuming from a queue are put by an administratively defined durable subscription, the application needs to access the user data information that is associated with the subscription. This information is added to the message as a property.

When a message is consumed from a queue that contains an RFH2 header with the MQPS folder, the value that is associated with the Sud key, if it exists, is added as a String property to the JMS Message object returned to the IBM MQ classes for JMS application. To enable the retrieval of this property from the message, the constant JMS_IBM_SUBSCRIPTION_USER_DATA in the JmsConstants interface can be used with the following method to get the subscription user data:

- JMS 3.0 jakarta.jms.Message.getStringProperty(java.lang.String)
- JMS 2.0 javax.jms.Message.getStringProperty(java.lang.String)

In the following example, an administrative durable subscription is defined by using the MQSC command **DEFINE SUB**:

```
DEFINE SUB('MY.SUBCRIPTION') TOPICSTR('PUBLIC') DEST('MY.SUBSCRIPTION.Q')
USERDATA('Administrative durable subscription to put message to the queue MY.SUBSCRIPTION.Q')
```

Copies of messages that are published to the topic string PUBLIC are put to the queue, MY.SUBSCRIPTION.Q. The user data that is associated with the durable subscription is then added as a property to the message, which is stored in the MQPS folder of the RFH2 header with the key Sud.

The IBM MQ classes for JMS application can call:

JMS 3.0
```
jakarta.jms.Message.getStringProperty(JmsConstants.JMS_IBM_SUBSCRIPTION_USER_DATA);
```

JMS 2.0
```
javax.jms.Message.getStringProperty(JmsConstants.JMS_IBM_SUBSCRIPTION_USER_DATA);
```

The following String is then returned:

```
Administrative durable subscription to put message to the queue MY.SUBSCRIPTION.Q
```

Related concepts

"The MQRFH2 header and JMS" on page 143

Related tasks

Defining an administrative subscription

Related reference

DEFINE SUB

Interface JmsConstants

Closing down an IBM MQ classes for JMS application

It is important for an IBM MQ classes for JMS application to close certain JMS objects explicitly before stopping. Finalizers might not be called, so do not rely on them to free resources. Do not allow an application to terminate with compressed trace active.

Garbage collection alone cannot release all IBM MQ classes for JMS and IBM MQ resources in a timely manner, especially if an application creates many short lived JMS objects at the session level or lower. It is therefore important for an application to close a Connection, Session, MessageConsumer, or MessageProducer object when it is no longer required.

If an application ends without closing a Connection, an implicit rollback occurs for all the connection's transacted sessions. To ensure any changes made by the application are committed, close the Connection explicitly before closing the application.

Do not use finalizers in an application to close JMS objects. Because finalizers might not be called, resources might not be freed. When a Connection is closed it closes all the Sessions that were created from it. Similarly, the MessageConsumers and MessageProducers created from a Session are closed when the Session is closed. However, consider closing Sessions, MessageConsumers, and MessageProducers explicitly to ensure resources are freed in a timely manner.

If trace compression is activated, System.Halt() shutdowns and abnormal, uncontrolled JVM terminations are likely to result in a corrupt trace file. Where possible, turn off the trace facility when you have collected the trace information you need. If you are tracing an application up to an abnormal end, use uncompressed trace output.

Note: To disconnect from a queue manager, a JMS application invokes the close() method on the connection object.

Handling poison messages in IBM MQ classes for JMS

A poison message is one which cannot be processed by a receiving application. If a poison message is delivered to an application and rolled back a specified number of times, the IBM MQ classes for JMS can move it to a backout queue.

A poison message is a message that cannot be processed by a receiving application. The message could have an unexpected type, or contain information that cannot be handled by the application's logic. If a poison message is delivered to an application, the application will be unable to process it and will roll it back to the queue where it came from. By default, the IBM MQ classes for JMS will repeatedly redeliver the message to the application. This can result in the application getting stuck in a loop continually trying to process the poison message and rolling it back.

To prevent this from happening, the IBM MQ classes for JMS can detect poison messages, and move them to an alternative destination. To do this, the IBM MQ classes for JMS make use of the following properties:

- The value of the BackoutCount field within the MQMD of the message that has been detected.
- The IBM MQ queue attributes **BOTHRESH** (backout threshold) and **BOQNAME** (backout requeue queue) for the input queue containing the message.

Whenever a message is rolled back by an application, the queue manager automatically increments the value of the BackoutCount field for the message.

When the IBM MQ classes for JMS detect a message that has a BackoutCount greater than zero, they compare the value of the BackoutCount to the value of the **BOTHRESH** attribute.

- If the BackoutCount is less than the value of the **BOTHRESH** attribute, the IBM MQ classes for JMS deliver it to the application for processing.
- However, if the BackoutCount is greater than or equal to **BOTHRESH**, then the message is considered to be a poison message. In this situation, the IBM MQ classes for JMS then move the message to the queue specified by the **BOQNAME** attribute. If the message cannot be put to the backout queue, then it is either moved to the queue manager's dead letter queue or discarded, depending upon the report options of the message.

Note:

- If the **BOTHRESH** attribute is left at its default value of `0`, then poison message handling is disabled. This means that any poison messages are put back to the input queue.
- The other thing to note is that IBM MQ classes for JMS query the **BOTHRESH** and **BOQNAME** attributes for the queue the first time they detect a message that has a BackoutCount greater than zero. The values of these attributes are then cached, and used whenever the IBM MQ classes for JMS encounter a message that has a BackoutCount greater than zero.

Configuring your system to perform poison message handling

The queue that the IBM MQ classes for JMS use when inquiring the **BOTHRESH** and **BOQNAME** attributes depends on the style of messaging being performed:

- For point-to-point messaging, this is the underlying local queue. This is important when a JMS application is consuming messages from either alias queues or cluster queues.
- For publish/subscribe messaging, a managed queue is created to hold the messages for an application. The IBM MQ classes for JMS query the managed queue to determine the values for the **BOTHRESH** and **BOQNAME** attributes.

  The managed queue is created from a model queue associated with the Topic object that the application has subscribed to, and inherits the values of the **BOTHRESH** and **BOQNAME** attributes from the model queue. The model queue that is used depends on whether the receiving application has taken out a durable or non-durable subscription:

  – The model queue used for durable subscriptions is specified by the **MDURMDL** attribute of the Topic. The default value of this attribute is `SYSTEM.DURABLE.MODEL.QUEUE`.
  – For non-durable subscriptions, the model queue that is used is specified by the **MNDURMDL** attribute. The default value of the **MNDURMDL** attribute is `SYSTEM.NDURABLE.MODEL.QUEUE`.

When inquiring the **BOTHRESH** and **BOQNAME** attributes, the IBM MQ classes for JMS:

- Open the local queue, or the target queue for an alias queue.
- Inquire the **BOTHRESH** and **BOQNAME** attributes.
- Close the local queue, or the target queue for an alias queue.

The open options that are used when opening the local queue, or the target queue for an alias queue, depend on the version of the IBM MQ classes for JMS being used:

- For IBM MQ classes for JMS in IBM MQ 9.1.0 Fix Pack 1 and earlier, or IBM MQ 9.1.1, if the local queue, or the target queue for an alias queue, is a cluster queue, then the IBM MQ classes for JMS open the queue with the `MQOO_INPUT_AS_Q_DEF`, `MQOO_INQUIRE` and `MQOO_FAIL_IF_QUIESCING` options. This means that the user running the receiving application must have inquire and get access to the local instance of the cluster queue.

  The IBM MQ classes for JMS open all other types of local queue with the open options `MQOO_INQUIRE` and `MQOO_FAIL_IF_QUIESCING`. In order for the IBM MQ classes for JMS to query the values of the attributes, the user running the receiving application must have inquire access on the local queue.
- When using the IBM MQ classes for JMS in IBM MQ 9.1.0 Fix Pack 2 and later, or for IBM MQ 9.1.2 and later, the user running the receiving application must have inquire access on the local queue, regardless of the type of the queue.

To move poison messages to either a backout requeue queue or the queue manager's dead letter queue, you must grant the user running the application `put` and `passall` authorities.

Processing poison messages for synchronous applications

If an application receives messages synchronously, by calling one of the following methods, the IBM MQ classes for JMS requeue a poison message within the unit of work that was active when the application tried to get the message:

- JMSConsumer.receive()
- JMSConsumer.receive(long timeout)
- JMSConsumer.receiveBody(Class<T> c)
- JMSConsumer.receiveBody(Class<T> c, long timeout)
- JMSConsumer.receiveBodyNoWait Class<T> c)
- JMSConsumer.receiveNoWait()
- MessageConsumer.receive()
- MessageConsumer.receive(long timeout)
- MessageConsumer.receiveNoWait()
- QueueReceiver.receive()

- QueueReceiver.receive(long timeout)
- QueueReceiver.receiveNoWait()
- TopicSubscriber.receive()
- TopicSubscriber.receive(long timeout)
- TopicSubscriber.receiveNoWait()

This means that if the application is using either a transacted JMS context or session, then the moving of the message to the backout queue is not committed until the transaction is committed.

If the **BOTHRESH** attribute is set to a value other than zero, the **BOQNAME** attribute should also be set. If the **BOTHRESH** is set to a value greater than zero, and the **BOQNAME** has not been set, the behavior is determined by the report options of the message:

- If the message has the report option `MQRO_DISCARD_MSG` set, the message is discarded.
- If the message has the report option `MQRO_DEAD_LETTER_Q` specified, then the IBM MQ classes for JMS try to move the message to the queue manager's dead letter queue.
- If the message does not have either the `MQRO_DISCARD_MSG` or `MQRO_DEAD_LETTER_Q` set, the IBM MQ classes for JMS try to put the message to the dead letter queue for the queue manager.

In the event that the attempt to put the message to the dead letter queue fails for some reason, what happens to the message is determined by whether the receiving application is using a transacted or non-transacted JMS context or session:

- If the receiving application is using either a transacted JMS context or session, and the transaction is committed, then the message is discarded.
- If the receiving application is using a transacted JMS context or session, and rolls the transaction back, the message is returned to the input queue.
- If the receiving application has created a non-transacted JMS context or session, the message is discarded.

Processing poison messages for asynchronous applications

If an application is receiving messages asynchronously via a MessageListener, the IBM MQ classes for JMS requeue poison messages without affecting message delivery. The requeue process takes place outside of any unit of work associated with actual message delivery to the application.

If the **BOTHRESH** is set to a value greater than zero, and the **BOQNAME** has not been set, the behavior is determined by the report options of the message:

- If the message has the report option `MQRO_DISCARD_MSG` set, the message is discarded.
- If the message has the report option `MQRO_DEAD_LETTER_Q` specified, then the IBM MQ classes for JMS try to move the message to the queue manager's dead letter queue.
- If the message does not have either the `MQRO_DISCARD_MSG` or `MQRO_DEAD_LETTER_Q` set, the IBM MQ classes for JMS try to put the message to the dead letter queue for the queue manager.

If the attempt to put the message to the dead letter queue fails for some reason, the IBM MQ classes for JMS returns the message to the input queue.

For information on how activation specifications and ConnectionConsumers handle poison messages, see Removing messages from the queue in ASF.

What happens to a message when it is moved to the backout queue

When a poison message is requeued to the backout requeue queue, the IBM MQ classes for JMS add an RFH2 header to it (if it did not have one already), and update some of the fields within the message descriptor (MQMD).

If the poison message contains an RFH2 header (because it was a JMS message, for example), the IBM MQ classes for JMS change the following fields within the MQMD when moving the message to the backout requeue queue:

- The BackoutCount field is reset to zero.
- The Expiry field of the message is updated to reflect the remaining expiry at the time the poison message was received by the JMS application.

If the poison message does not contain an RFH2 header, the IBM MQ classes for JMS add one and update the following fields in the MQMD as part of the backout processing:

- The BackoutCount field is reset to zero.
- The Expiry field of the message is updated to reflect the remaining expiry at the time the poison message was received by the JMS application.
- The Format field of the message is changed to MQHRF2.
- The CCSID field is changed to be 1208.
- The Encoding field is modified to be 273.

In addition to this, the CCSID and Encoding fields from the poison message are copied into the CCSID and Encoding fields of the RFH2 header, to ensure that the header chaining of the message on the backout requeue queue is correct.

Related concepts

"Handling poison messages in ASF" on page 325
Within the Application Server Facilities, poison message handling is handled slightly differently to elsewhere in IBM MQ classes for JMS.

Exceptions in IBM MQ classes for JMS

An IBM MQ classes for JMS application must handle exceptions that are thrown by JMS API calls or that are delivered to an exception handler.

IBM MQ classes for JMS reports runtime problems by throwing exceptions. The type of exceptions that are thrown, and the way that these exceptions must be handled, depends on the version of the JMS specification that is used by your application:

- Methods on the interfaces that are defined in JMS 1.1 and earlier throw checked exceptions. The base class of these exceptions is `JMSException`. For more information about how to handle checked exceptions, see "Handling checked exceptions" on page 228.
- Methods on the interfaces added in JMS 2.0 throw unchecked exceptions. The base class for these exceptions is `JMSRuntimeException`. For more information on how to handle unchecked exceptions, see "Handling unchecked exceptions" on page 232.

You can also register an `ExceptionListener` with a JMS `Connection` or a `JMSContext`. The MQ classes for JMS then notify the `ExceptionListener` if either an issue is detected with a connection to the queue manager, or if an issue occurs while trying to deliver a message asynchronously. For more information, see "ExceptionListeners" on page 235.

Related concepts

IBM MQ classes for JMS

Related reference

ASYNCEXCEPTION

Handling checked exceptions

Methods on the interfaces that are defined in JMS 1.1 or earlier throw checked exceptions. The base class for these exceptions is `JMSException`. Therefore, catching `JMSExceptions` provides a generic way of handling these types of exceptions.

Every `JMSException` encapsulates the following information:

- A provider specific exception message, which your application can obtain by calling the `Throwable.getMessage()` method.
- A provider specific error code, which your application can obtain by calling the `JMSException.getErrorCode()` method.
- A linked exception. An exception that is thrown by a JMS 1.1 API call is often the result of a lower level problem which is reported by another exception that is linked to this exception. Your application can obtain a linked exception by calling either the `JMSException.getLinkedException()` method or the `Throwable.getCause()` method.

When you use the JMS 1.1 API, most exceptions that are thrown by the IBM MQ classes for JMS are instances of subclasses of `JMSException`. These subclasses implement the `com.ibm.msg.client.jms.JmsExceptionDetail` interface, which provides the following additional information:

- An explanation of the exception message. Your application can obtain this message by calling the `JmsExceptionDetail.getExplanation()` method.
- A recommended user response to the exception. Your application can obtain this message by calling the `JmsExceptionDetail.getUserAction()` method.
- The keys for the message inserts in the exception message. Your application can obtain an iterator for all the keys by calling the `JmsExceptionDetail.getKeys()` method.
- The message inserts in the exception message. For example, a message insert might be the name of the queue that caused the exception, and it might be useful for your application to access that name. Your application can obtain the message insert corresponding to a specified key by calling the `JmsExceptionDetail.getValue()` method.

All the methods in the `JmsExceptionDetail` interface return null if no details are available.

For example, if an application tries to create a message producer for an IBM MQ queue that does not exist, an exception is thrown with the following information:

```
Message : JMSWMQ2008: Failed to open MQ queue 'Q_test'.
Class : class com.ibm.msg.client.jms.DetailedInvalidDestinationException
Error Code : JMSWMQ2008
Explanation : JMS attempted to perform an MQOPEN, but IBM MQ reported an
              error.
User Action : Use the linked exception to determine the cause of this error. Check
              that the specified queue and queue manager are defined correctly.
```

The exception that is thrown, `com.ibm.msg.client.jms.DetailedInvalidDestinationException`, is a subclass of the following class, and implements the `com.ibm.msg.client.jms.JmsExceptionDetail` interface.

- JMS 3.0 `jakarta.jms.InvalidDestinationException`
- JMS 2.0 `javax.jms.InvalidDestinationException`

Linked exceptions

A linked exception provides further information about a runtime problem. Therefore, for each `JMSException` that is thrown, an application should check the linked exception.

The linked exception itself might have another linked exception, and so the linked exceptions form a chain that leads back to the original underlying problem. A linked exception is implemented by using the chained exception mechanism of the `java.lang.Throwable` class, and your application can obtain a linked exception by calling the `Throwable.getCause()` method. For a `JMSException`, the `getLinkedException()` method delegates to the `Throwable.getCause()` method.

For example, if an application specifies an incorrect port number when connecting to a queue manager, the exceptions form the following chain:

```
com.ibm.msg.client.jms.DetailIllegalStateException
|
+--->
    com.ibm.mq.MQException
    |
    +--->
        com.ibm.mq.jmqi.JmqiException
        |
        +--->
            com.ibm.mq.jmqi.JmqiException
            |
            +--->
                java.net.ConnectionException
```

Typically, each exception in a chain is thrown from a different layer in the code. For example, the exceptions in the preceding chain are thrown by the following layers:

- The first exception, an instance of a subclass of `JMSException`, is thrown by the common layer in IBM MQ classes for JMS.
- The next exception, an instance of `com.ibm.mq.MQException`, is thrown by the IBM MQ messaging provider.
- The next two exceptions, both of which are instances of `com.ibm.mq.jmqi.JmqiException`, are thrown by the Java Message Queueing Interface (JMQI). The JMQI is the component that is used by the IBM MQ classes for JMS to communicate with a queue manager.
- The final exception, an instance of `java.net.ConnectionException`, is thrown by the Java class library.

For more information about the layered architecture of IBM MQ classes for JMS, see IBM MQ classes for JMS architecture.

You can code your application to iterate through this chain to extract all the appropriate information, as shown in the following example:

JMS 3.0

```
import com.ibm.msg.client.jms.JmsExceptionDetail;
import com.ibm.mq.MQException;
import com.ibm.mq.jmqi.JmqiException;
import jakarta.jms.JMSException;
.
.
.
catch (JMSException je) {
  System.err.println("Caught JMSException");
  // Check for linked exceptions in JMSException
  Throwable t = je;
  while (t != null) {
    // Write out the message that is applicable to all exceptions
    System.err.println("Exception Msg: " + t.getMessage());
    // Write out the exception stack trace
    t.printStackTrace(System.err);

    // Add on specific information depending on the type of exception
    if (t instanceof JMSException) {
      JMSException je1 = (JMSException) t;
      System.err.println("JMS Error code: " + je1.getErrorCode());
      if (t instanceof JmsExceptionDetail){
        JmsExceptionDetail jed = (JmsExceptionDetail)je1;
        System.err.println("JMS Explanation: " + jed.getExplanation());
        System.err.println("JMS Explanation: " + jed.getUserAction());
      }
    } else if (t instanceof MQException) {
      MQException mqe = (MQException) t;
      System.err.println("WMQ Completion code: " + mqe.getCompCode());
      System.err.println("WMQ Reason code: " + mqe.getReason());
    } else if (t instanceof JmqiException){
      JmqiException jmqie = (JmqiException)t;
      System.err.println("WMQ Log Message: " + jmqie.getWmqLogMessage());
      System.err.println("WMQ Explanation: " + jmqie.getWmqMsgExplanation());
      System.err.println("WMQ Msg Summary: " + jmqie.getWmqMsgSummary());
      System.err.println("WMQ Msg User Response: " + jmqie.getWmqMsgUserResponse());
      System.err.println("WMQ Msg Severity: " + jmqie.getWmqMsgSeverity());
    }
```

```
    // Get the next cause
    t = t.getCause();
  }
}
```

JMS 2.0

```
import com.ibm.msg.client.jms.JmsExceptionDetail;
import com.ibm.mq.MQException;
import com.ibm.mq.jmqi.JmqiException;
import javax.jms.JMSException;
.
.
.
catch (JMSException je) {
  System.err.println("Caught JMSException");
  // Check for linked exceptions in JMSException
  Throwable t = je;
  while (t != null) {
    // Write out the message that is applicable to all exceptions
    System.err.println("Exception Msg: " + t.getMessage());
    // Write out the exception stack trace
    t.printStackTrace(System.err);

    // Add on specific information depending on the type of exception
    if (t instanceof JMSException) {
      JMSException je1 = (JMSException) t;
      System.err.println("JMS Error code: " + je1.getErrorCode());
      if (t instanceof JmsExceptionDetail){
        JmsExceptionDetail jed = (JmsExceptionDetail)je1;
        System.err.println("JMS Explanation: " + jed.getExplanation());
        System.err.println("JMS Explanation: " + jed.getUserAction());
      }
    } else if (t instanceof MQException) {
      MQException mqe = (MQException) t;
      System.err.println("WMQ Completion code: " + mqe.getCompCode());
      System.err.println("WMQ Reason code: " + mqe.getReason());
    } else if (t instanceof JmqiException){
      JmqiException jmqie = (JmqiException)t;
      System.err.println("WMQ Log Message: " + jmqie.getWmqLogMessage());
      System.err.println("WMQ Explanation: " + jmqie.getWmqMsgExplanation());
      System.err.println("WMQ Msg Summary: " + jmqie.getWmqMsgSummary());
      System.err.println("WMQ Msg User Response: " + jmqie.getWmqMsgUserResponse());
      System.err.println("WMQ Msg Severity: " + jmqie.getWmqMsgSeverity());
    }
    // Get the next cause
    t = t.getCause();
  }
}
```

Note that your application should always check the type of each exception in a chain because the type of exception can vary and exceptions of different types encapsulate different information.

Obtaining IBM MQ specific information about a problem

Instances of `com.ibm.mq.MQException` and `com.ibm.mq.jmqi.JmqiException` encapsulate IBM MQ specific information about a problem.

An `MQException` encapsulates the following information:

- A completion code, which your application can obtain by calling the `getCompCode()` method.
- A reason code, which your application can obtain by calling the `getReason()` method.

For examples of how to use these methods, see the sample code in linked exceptions.

A `JmqiException` also encapsulates a completion code and a reason code. In addition to this, a `JmqiException` contains the information in an AMQ *nnnn* or CSQ *nnnn* message, if one is associated with the exception. Your application can obtain the various components of this message by calling the following methods:

- The `getWmqMsgExplanation()` method returns the explanation of the AMQ *nnnn* or CSQ *nnnn* message.
- The `getWmqMsgSeverity()` method returns the severity of the AMQ *nnnn* or CSQ *nnnn* message.

- The `getWmqMsgSummary()` method returns the summary of the AMQ *nnnn* or CSQ *nnnn* message.
- The `getWmqMsgUserResponse()` method returns the user response that is associated with the AMQ *nnnn* or CSQ *nnnn* message.

Handling unchecked exceptions
Methods on the interfaces that are defined in JMS 2.0 throw unchecked exceptions. The base class for these exceptions is `JMSRuntimeException`. Therefore, catching `JMSRuntimeExceptions` provides a generic way of handling these types of exceptions.

Every `JMSRuntimeException` encapsulates the following information:

- A provider specific exception message, which your application can obtain by calling the `JMSRuntimeException.getMessage()` method.
- A provider specific error code, which your application can obtain by calling the `JMSRuntimeException.getErrorCode()` method.
- A linked exception. An exception that is thrown by a JMS 2.0 API call is often the result of a lower level problem which is reported by another exception that is linked to this exception. Your application can obtain a linked exception by calling the `JMSRuntimeException.getCause()` method.

When you call methods on the interfaces that are provided by the JMS 2.0 API, most exceptions that are thrown by the IBM MQ classes for JMS are instances of subclasses of `JMSRuntimeException`. These subclasses implement the `com.ibm.msg.client.jms.JmsExceptionDetail` interface, which provides the following additional information:

- An explanation of the exception message. Your application can obtain this message by calling the `JmsExceptionDetail.getExplanation()` method.
- A recommended user response to the exception. Your application can obtain this message by calling the `JmsExceptionDetail.getUserAction()` method.
- The keys for the message inserts in the exception message. Your application can obtain an iterator for all the keys by calling the `JmsExceptionDetail.getKeys()` method.
- The message inserts in the exception message. For example, a message insert might be the name of the queue that caused the exception, and it might be useful for your application to access that name. Your application can obtain the message insert corresponding to a specified key by calling the `JmsExceptionDetail.getValue()` method.

All the methods in the `JmsExceptionDetail` interface return null if no details are available.

For example, if an application tries to create a `JMSProducer` for an IBM MQ queue that does not exist, an exception is thrown with the following information:

```
Message : JMSWMQ2008: Failed to open MQ queue 'Q_test'.
Class : class com.ibm.msg.client.jms.DetailedInvalidDestinationException
Error Code : JMSWMQ2008
Explanation : JMS attempted to perform an MQOPEN, but IBM MQ reported an
              error.
User Action : Use the linked exception to determine the cause of this error. Check
              that the specified queue and queue manager are defined correctly.
```

The exception that is thrown, `com.ibm.msg.client.jms.DetailedInvalidDestinationException`, is a subclass of the following class, and implements the `com.ibm.msg.client.jms.JmsExceptionDetail` interface.

- JMS 3.0 `jakarta.jms.InvalidDestinationException`
- JMS 2.0 `javax.jms.InvalidDestinationException`

Chained exceptions

Typically, exceptions are caused by other exceptions. Therefore, for each `JMSRuntimeException` that is thrown, your application should check the linked exception.

The cause of the JMSRuntimeException might be another exception. These exceptions form a chain that leads back to the original underlying problem. The cause of an exception is implemented by using the chained exception mechanism of the `java.lang.Throwable` class, and your application can obtain a linked exception by calling the `Throwable.getCause()` method.

For example, if an application specifies an incorrect port number when connecting to a queue manager, the exceptions form the following chain:

```
com.ibm.msg.client.jms.DetailIllegalStateException
|
+--->
    com.ibm.mq.MQException
    |
    +--->
        com.ibm.mq.jmqi.JmqiException
        |
        +--->
            com.ibm.mq.jmqi.JmqiException
            |
            +--->
                java.net.ConnectionException
```

Typically, each exception in a chain is thrown from a different layer in the code. For example, the exceptions in the preceding chain are thrown by the following layers:

- The first exception, an instance of a subclass of JMSRuntimeException, is thrown by the common layer in IBM MQ classes for JMS.
- The next exception, an instance of `com.ibm.mq.MQException`, is thrown by the IBM MQ messaging provider.
- The next two exceptions, both of which are instances of `com.ibm.mq.jmqi.JmqiException`, are thrown by the Java Message Queueing Interface (JMQI). The JMQI is the component that is used by the IBM MQ classes for JMS to communicate with a queue manager.
- The final exception, an instance of `java.net.ConnectionException`, is thrown by the Java class library.

For more information about the layered architecture of IBM MQ classes for JMS, see IBM MQ classes for JMS architecture.

You can code your application to iterate through this chain to extract all the appropriate information, as shown in the following example:

JMS 3.0

```
import com.ibm.msg.client.jms.JmsExceptionDetail;
import com.ibm.mq.MQException;
import com.ibm.mq.jmqi.JmqiException;
import jakarta.jms.JMSRuntimeException;
.
.
.
catch (JMSRuntimeException je) {
  System.err.println("Caught JMSRuntimeException");
  // Check for linked exceptions in JMSRuntimeException
  Throwable t = je;
  while (t != null) {
    // Write out the message that is applicable to all exceptions
    System.err.println("Exception Msg: " + t.getMessage());
    // Write out the exception stack trace
    t.printStackTrace(System.err);

    // Add on specific information depending on the type of exception
    if (t instanceof JMSRuntimeException) {
      JMSRuntimeException je1 = (JMSRuntimeException) t;
      System.err.println("JMS Error code: " + je1.getErrorCode());
      if (t instanceof JmsExceptionDetail){
        JmsExceptionDetail jed = (JmsExceptionDetail)je1;
        System.err.println("JMS Explanation: " + jed.getExplanation());
        System.err.println("JMS Explanation: " + jed.getUserAction());
      }
    } else if (t instanceof MQException) {
      MQException mqe = (MQException) t;
```

```
      System.err.println("WMQ Completion code: " + mqe.getCompCode());
      System.err.println("WMQ Reason code: " + mqe.getReason());
    } else if (t instanceof JmqiException){
      JmqiException jmqie = (JmqiException)t;
      System.err.println("WMQ Log Message: " + jmqie.getWmqLogMessage());
      System.err.println("WMQ Explanation: " + jmqie.getWmqMsgExplanation());
      System.err.println("WMQ Msg Summary: " + jmqie.getWmqMsgSummary());
      System.err.println("WMQ Msg User Response: " + jmqie.getWmqMsgUserResponse());
      System.err.println("WMQ Msg Severity: " + jmqie.getWmqMsgSeverity());
    }
    // Get the next cause
    t = t.getCause();
  }
}
```

JMS 2.0

```
import com.ibm.msg.client.jms.JmsExceptionDetail;
import com.ibm.mq.MQException;
import com.ibm.mq.jmqi.JmqiException;
import javax.jms.JMSRuntimeException;
.
.
.
catch (JMSRuntimeException je) {
  System.err.println("Caught JMSRuntimeException");
  // Check for linked exceptions in JMSRuntimeException
  Throwable t = je;
  while (t != null) {
    // Write out the message that is applicable to all exceptions
    System.err.println("Exception Msg: " + t.getMessage());
    // Write out the exception stack trace
    t.printStackTrace(System.err);

    // Add on specific information depending on the type of exception
    if (t instanceof JMSRuntimeException) {
      JMSRuntimeException je1 = (JMSRuntimeException) t;
      System.err.println("JMS Error code: " + je1.getErrorCode());
      if (t instanceof JmsExceptionDetail){
        JmsExceptionDetail jed = (JmsExceptionDetail)je1;
        System.err.println("JMS Explanation: " + jed.getExplanation());
        System.err.println("JMS Explanation: " + jed.getUserAction());
      }
    } else if (t instanceof MQException) {
      MQException mqe = (MQException) t;
      System.err.println("WMQ Completion code: " + mqe.getCompCode());
      System.err.println("WMQ Reason code: " + mqe.getReason());
    } else if (t instanceof JmqiException){
      JmqiException jmqie = (JmqiException)t;
      System.err.println("WMQ Log Message: " + jmqie.getWmqLogMessage());
      System.err.println("WMQ Explanation: " + jmqie.getWmqMsgExplanation());
      System.err.println("WMQ Msg Summary: " + jmqie.getWmqMsgSummary());
      System.err.println("WMQ Msg User Response: " + jmqie.getWmqMsgUserResponse());
      System.err.println("WMQ Msg Severity: " + jmqie.getWmqMsgSeverity());
    }
    // Get the next cause
    t = t.getCause();
  }
}
```

Note that your application should always check the type of each exception in a chain because the type of exception can vary and exceptions of different types encapsulate different information.

Obtaining IBM MQ specific information about a problem

Instances of `com.ibm.mq.MQException` and `com.ibm.mq.jmqi.JmqiException` encapsulate IBM MQ specific information about a problem.

An `MQException` encapsulates the following information:

- A completion code, which your application can obtain by calling the `getCompCode()` method.
- A reason code, which your application can obtain by calling the `getReason()` method.

For examples of how to use these methods, see the sample code in chained exceptions.

A JmqiException also encapsulates a completion code and a reason code. In addition to this, a JmqiException contains the information in an AMQ *nnnn* or CSQ *nnnn* message, if one is associated with the exception. Your application can obtain the various components of this message by calling the following methods:

- The getWmqMsgExplanation() method returns the explanation of the AMQ *nnnn* or CSQ *nnnn* message.
- The getWmqMsgSeverity() method returns the severity of the AMQ *nnnn* or CSQ *nnnn* message.
- The getWmqMsgSummary() method returns the summary of the AMQ *nnnn* or CSQ *nnnn* message.
- The getWmqMsgUserResponse() method returns the user response that is associated with the AMQ *nnnn* or CSQ *nnnn* message.

ExceptionListeners
JMS Connection and JMSContext objects have an associated connection to a queue manager. Your application can register an ExceptionListener with a JMS Connection or JMSContext. If a problem occurs that makes the connection that is associated with the Connection or JMSContext unusable, the IBM MQ classes for JMS delivers an exception to the ExceptionListener by calling its onException() method. Your application then has the opportunity to reestablish the connection.

IBM MQ classes for JMS can also deliver an exception to the exception listener if a problem occurs while trying to deliver a message asynchronously.

Exception listeners

From IBM MQ 8.0.0 Fix Pack 2, to maintain behavior for current JMS applications that configure a JMS MessageListener and a JMS ExceptionListener, and to ensure that the IBM MQ classes for JMS are consistent with the JMS specification, the default value for the ConnectionFactory property ASYNCEXCEPTION is changed to ASYNC_EXCEPTIONS_CONNECTIONBROKEN. As a result, only exceptions that correspond to broken connection error codes are delivered to the ExceptionListener of an application.

APAR IT14820, included from IBM MQ 9.0.0 Fix Pack 1, updates IBM MQ classes for JMS so that:

- An ExceptionListener that is registered by an application is invoked for any connection broken exceptions, regardless of whether the application is using synchronous or asynchronous message consumers.
- Non-connection broken exceptions (for example MQRC_GET_INHIBITED) that arise during message delivery are delivered to the ExceptionListener of an application when the application is using asynchronous message consumers and the JMS ConnectionFactory that is used by the application has the ASYNC_EXCEPTION property set to the value ASYNC_EXCEPTIONS_ALL.

Note: An ExceptionListener is invoked only once for a connection broken exception, even if two TCP/IP connections (one used by a JMS Connection and one used by a JMS Session) are broken.

For any other type of problem, an exception is thrown by the current JMS API call. The type of exception that is thrown depends on the version of the JMS API that the application is using:

- If the application is using the interfaces that are provided by the JMS 1.1 specification, the exception is a JMSException. For more information on how to handle these exceptions, see "Handling checked exceptions" on page 228.
- If the application is using JMS 2.0 interfaces, the exception is a JMSRuntimeException. For more information on how to handle these exceptions, see "Handling unchecked exceptions" on page 232.

If an application does not register an exception listener with a Connection or JMSContext, any exceptions that would be delivered to the exception listener are written to the IBM MQ classes for JMS log.

Accessing IBM MQ features from an IBM MQ classes for JMS application

IBM MQ classes for JMS provides facilities to exploit a number of features of IBM MQ.

Attention: These features are outside the JMS specification or, in certain cases, violate the JMS specification. If you use them, your application is unlikely to be compatible with other JMS providers. Those features which do not comply with the JMS specification are labeled with an Attention notice.

Reading and writing the message descriptor from an IBM MQ classes for JMS application

You control the ability to access the message descriptor (MQMD) by setting properties on a Destination and a Message.

Some IBM MQ applications require specific values to be set in the MQMD of messages sent to them. IBM MQ classes for JMS provides message attributes that allow JMS applications to set MQMD fields and so enable JMS applications to "drive" IBM MQ applications.

You must set the Destination object property WMQ_MQMD_WRITE_ENABLED to true for the setting of MQMD properties to have any effect. You can then use the property setting methods of the message (for example setStringProperty) to assign values to the MQMD fields. All MQMD fields are exposed except StrucId and Version; BackoutCount can be read but not written to.

This example results in a message being put to a queue or topic with MQMD.UserIdentifier set to "JoeBloggs".

```
// Create a ConnectionFactory, connection, session, producer, message
// ...

// Create a destination
// ...

// Enable MQMD write
dest.setBooleanProperty(WMQConstants.WMQ_MQMD_WRITE_ENABLED, true);

// Optionally, set a message context if applicable for this MD field
dest.setIntProperty(WMQConstants.WMQ_MQMD_MESSAGE_CONTEXT,
  WMQConstants.WMQ_MDCTX_SET_IDENTITY_CONTEXT);

// On the message, set property to provide custom UserId
msg.setStringProperty("JMS_IBM_MQMD_UserIdentifier", "JoeBloggs");

// Send the message
// ...
```

It is necessary to set WMQ_MQMD_MESSAGE_CONTEXT before setting JMS_IBM_MQMD_UserIdentifier. For more information about the use of WMQ_MQMD_MESSAGE_CONTEXT, see “JMS message object properties” on page 239.

Similarly, you can extract the contents of the MQMD fields by setting WMQ_MQMD_READ_ENABLED to true before receiving a message and then using the get methods of the message, such as getStringProperty. Any properties received are read-only.

This example results in the *value* field holding the value of the MQMD.ApplIdentityData field of a message got from a queue or a topic.

```
// Create a ConnectionFactory, connection, session, consumer
// ...

// Create a destination
// ...

// Enable MQMD read
dest.setBooleanProperty(WMQConstants.WMQ_MQMD_READ_ENABLED, true);

// Receive a message
// ...

// Get MQMD field value using a property
String value = rcvMsg.getStringProperty("JMS_IBM_MQMD_ApplIdentityData");
```

JMS destination object properties
Two properties of the Destination object control access to the MQMD from JMS, and a third controls message context.

Table 36. Property names and descriptions

Property	Short form	Description
WMQ_MQMD_WRITE_ENABLED	MDW	Whether a JMS application can set the values of MQMD fields
WMQ_MQMD_READ_ENABLED	MDR	Whether a JMS application can extract the values of MQMD fields
WMQ_MQMD_MESSAGE_ CONTEXT	MDCTX	What level of message context is to be set by the JMS application. The application must be running with appropriate context authority for this property to take effect

Table 37. Property names, values, and set methods

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MQMD_WRITE _ENABLED	• **NO** All JMS_IBM_MQMD* properties are ignored and their values are not copied into the underlying MQMD structure. • YES JMS_IBM_MQMD* properties are processed. Their values are copied into the underlying MQMD structure.	• **False** • True	setMQMDWriteEnabled

Table 37. Property names, values, and set methods (continued)

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MQMD_READ_ENABLED	• **NO** When sending messages, the JMS_IBM_MQMD* properties on a sent message are not updated to reflect the updated field values in the MQMD. When receiving messages, none of the JMS_IBM_MQMD* properties are available on a received message, even if the sender had set some or all of them. • YES When sending messages, all of the JMS_IBM_MQMD* properties on a sent message are updated to reflect the updated field values in the MQMD, including those that the sender did not set explicitly. When receiving messages, all of the JMS_IBM_MQMD* properties are available on a received message, including those that the sender did not set explicitly.	• **False** • True	setMQMDReadEnabled
WMQ_MQMD_MESSAGE_CONTEXT	• **DEFAULT** The MQOPEN API call and the MQPMO structure specify no explicit message context options • SET_IDENTITY_CONTEXT The MQOPEN API call specifies the message context option MQOO_SET_IDENTITY_CONTEXT and the MQPMO structure specifies MQPMO_SET_IDENTITY_CONTEXT • SET_ALL_CONTEXT The MQOPEN API call specifies the message context option MQOO_SET_ALL_CONTEXT and the MQPMO structure specifies MQPMO_SET_ALL_CONTEXT	• **WMQ_MDCTX_DEFAULT** • WMQ_MDCTX_SET_IDENTITY_CONTEXT • WMQ_MDCTX_SET_ALL_CONTEXT	setMQMDMessageContext

JMS message object properties
Message object properties prefixed JMS_IBM_MQMD allow you to set or read the corresponding MQMD field.

Sending messages

All MQMD fields except StrucId and Version are represented. These properties refer only to the MQMD fields; where a property occurs both in the MQMD and in the MQRFH2 header, the version in the MQRFH2 is not set or extracted.

Any of these properties can be set, except JMS_IBM_MQMD_BackoutCount. Any value set for JMS_IBM_MQMD_BackoutCount is ignored.

If a property has a maximum length and you supply a value that is too long, the value is truncated.

For certain properties, you must also set the WMQ_MQMD_MESSAGE_CONTEXT property on the Destination object. The application must be running with appropriate context authority for this property to take effect. If you do not set WMQ_MQMD_MESSAGE_CONTEXT to an appropriate value, the property value is ignored. If you set WMQ_MQMD_MESSAGE_CONTEXT to an appropriate value but you do not have sufficient context authority for the queue manager, a JMSException is issued. Properties requiring specific values of WMQ_MQMD_MESSAGE_CONTEXT are as follows.

The following properties require WMQ_MQMD_MESSAGE_CONTEXT to be set to WMQ_MDCTX_SET_IDENTITY_CONTEXT or WMQ_MDCTX_SET_ALL_CONTEXT:

- JMS_IBM_MQMD_UserIdentifier
- JMS_IBM_MQMD_AccountingToken
- JMS_IBM_MQMD_ApplIdentityData

The following properties require WMQ_MQMD_MESSAGE_CONTEXT to be set to WMQ_MDCTX_SET_ALL_CONTEXT :

- JMS_IBM_MQMD_PutApplType
- JMS_IBM_MQMD_PutApplName
- JMS_IBM_MQMD_PutDate
- JMS_IBM_MQMD_PutTime
- JMS_IBM_MQMD_ApplOriginData

Receiving messages

All these properties are available on a received message if WMQ_MQMD_READ_ENABLED property is set to true, irrespective of the actual properties the producing application has set. An application cannot modify the properties of a received message unless all properties are cleared first, according to the JMS specification. The received message can be forwarded without modifying the properties.

Attention: If your application receives a message from a destination with WMQ_MQMD_READ_ENABLED property set to true, and forwards it to a destination with WMQ_MQMD_WRITE_ENABLED set to true, this results in all the MQMD field values of the received message being copied into the forwarded message.

Table of properties

This table lists the properties of the Message object representing the MQMD fields. See the links for full descriptions of the fields and their allowable values.

Table 38. Property names, descriptions, and types

Property	Description	Java Type	Link to full description
JMS_IBM_MQMD_Report	Options for report messages	Integer	Report

Table 38. Property names, descriptions, and types (continued)

Property	Description	Java Type	Link to full description
JMS_IBM_MQMD_MsgType	Message type	Integer	MsgType
JMS_IBM_MQMD_Expiry	Message lifetime	Integer	Expiry
JMS_IBM_MQMD_Feedback	Feedback or reason code	Integer	Feedback
JMS_IBM_MQMD_Encoding	Numeric encoding of message data	Integer	Encoding
JMS_IBM_MQMD_CodedCharSetId	Character set identifier of message data	Integer	CodedCharSetId
JMS_IBM_MQMD_Format	Format name of message data	String	Format
JMS_IBM_MQMD_Priority [1]	Message priority	Integer	Priority
JMS_IBM_MQMD_Persistence	Message persistence	Integer	Persistence
JMS_IBM_MQMD_MsgId [2]	Message identifier	Object (byte[]) [4]	MsgId
JMS_IBM_MQMD_CorrelId [3]	Correlation identifier	Object (byte[]) [4]	CorrelId
JMS_IBM_MQMD_BackoutCount	Backout counter	Integer	BackoutCount
JMS_IBM_MQMD_ReplyToQ	Name of reply queue	String	ReplyToQ
JMS_IBM_MQMD_ReplyToQMgr	Name of reply queue manager	String	ReplyToQMgr
JMS_IBM_MQMD_UserIdentifier	User identifier	String	UserIdentifier
JMS_IBM_MQMD_AccountingToken	Accounting token	Object (byte[]) [4]	AccountingToken
JMS_IBM_MQMD_ApplIdentityData	Application data relating to identity	String	ApplIdentityData
JMS_IBM_MQMD_PutApplType	Type of application that put the message	Integer	PutApplType
JMS_IBM_MQMD_PutApplName	Name of application that put the message	String	PutApplName
JMS_IBM_MQMD_PutDate	Date when message was put	String	PutDate
JMS_IBM_MQMD_PutTime	Time when message was put	String	PutTime
JMS_IBM_MQMD_ApplOriginData	Application data relating to origin	String	ApplOriginData
JMS_IBM_MQMD_GroupId	Group identifier	Object (byte[]) [4]	GroupId
JMS_IBM_MQMD_MsgSeqNumber	Sequence number of logical message within group	Integer	MsgSeqNumber

Table 38. Property names, descriptions, and types (continued)			
Property	**Description**	**Java Type**	**Link to full description**
JMS_IBM_MQMD_Offset	Offset of data in physical message from start of logical message	Integer	Offset
JMS_IBM_MQMD_MsgFlags	Message flags	Integer	MsgFlags
JMS_IBM_MQMD_OriginalLength	Length of original message	Integer	OriginalLength

1. **Attention:** If you assign a value to JMS_IBM_MQMD_Priority that is not within the range 0-9, this violates the JMS specification.
2. **Attention:** The JMS specification states that the message ID must be set by the JMS provider and that it must either be unique or null. If you assign a value to JMS_IBM_MQMD_MsgId, this value is copied to the JMSMessageID. Thus it is not set by the JMS provider and might not be unique: this violates the JMS specification.
3. **Attention:** If you assign a value to JMS_IBM_MQMD_CorrelId that starts with the string 'ID:', this violates the JMS specification.
4. **Attention:** The use of byte array properties on a message violates the JMS specification.

Accessing IBM MQ Message data from an application using IBM MQ classes for JMS
You can access the complete IBM MQ message data within an application using IBM MQ classes for JMS. To access all the data, the message must be a `JMSBytesMessage`. The body of the `JMSBytesMessage` includes any MQRFH2 header, any other IBM MQ headers, and the following message data.

Set the `WMQ_MESSAGE_BODY` property of the destination to `WMQ_MESSAGE_BODY_MQ`, to receive all the message body data in the `JMSBytesMessage`.

If `WMQ_MESSAGE_BODY` is set to `WMQ_MESSAGE_BODY_JMS` or `WMQ_MESSAGE_BODY_UNSPECIFIED`, the message body is returned without the JMS MQRFH2 header, and the properties of the `JMSBytesMessage` reflect the properties set in the RFH2.

Some applications cannot use the functions described in this topic. If an application is connected to an IBM MQ V6 queue manager, or if it has set `PROVIDERVERSION` to 6, the functions are not available.

Sending a message

When sending messages the destination property, `WMQ_MESSAGE_BODY`, takes precedence over `WMQ_TARGET_CLIENT`.

If `WMQ_MESSAGE_BODY` is set to `WMQ_MESSAGE_BODY_JMS`, IBM MQ classes for JMS automatically generates an MQRFH2 header based on the settings of the `JMSMessage` properties and header fields.

If `WMQ_MESSAGE_BODY` is set to `WMQ_MESSAGE_BODY_MQ`, no additional header is added to the message body

If `WMQ_MESSAGE_BODY` is set to `WMQ_MESSAGE_BODY_UNSPECIFIED`, IBM MQ classes for JMS sends an MQRFH2 header, unless `WMQ_TARGET_CLIENT` is set to `WMQ_TARGET_DEST_MQ`. On receive, setting `WMQ_TARGET_CLIENT` to `WMQ_TARGET_DEST_MQ` results in any MQRFH2 being removed from the message body.

Note: `JMSBytesMessage` and `JMSTextMessage` do not require an MQRFH2, whereas `JMSStreamMessage`, `JMSMapMessage`, and `JMSObjectMessage` do.

`WMQ_MESSAGE_BODY_UNSPECIFIED` is the default setting for `WMQ_MESSAGE_BODY`, and `WMQ_TARGET_DEST_JMS` is the default setting for `WMQ_TARGET_CLIENT`.

If you send a JMSBytesMessage, you can override the default settings for the JMS message body when the IBM MQ message is constructed. Use the following properties:

- JMS_IBM_Format or JMS_IBM_MQMD_Format: This property specifies the format of the IBM MQ header or application payload that starts the JMS message body if there is no preceding WebSphere MQ header.
- JMS_IBM_Character_Set or JMS_IBM_MQMD_CodedCharSetId: This property specifies the CCSID of the IBM MQ header or application payload that starts the JMS message body if there is no preceding WebSphere MQ header.
- JMS_IBM_Encoding or JMS_IBM_MQMD_Encoding: This property specifies the encoding of the IBM MQ header or application payload that starts the JMS message body if there is no preceding WebSphere MQ header.

If both types of property are specified, the JMS_IBM_MQMD_* properties override the corresponding JMS_IBM_* properties, as long as the destination property WMQ_MQMD_WRITE_ENABLED is set to true.

The differences in effect between setting message properties using JMS_IBM_MQMD_* and JMS_IBM_* are significant:

1. The JMS_IBM_MQMD_* properties are specific to the IBM MQ JMS provider.
2. The JMS_IBM_MQMD_* properties are only set in the MQMD. JMS_IBM_* properties are set in the MQMD only if the message does not have an MQRFH2 JMS header. Otherwise they are set in the JMS RFH2 header.
3. The JMS_IBM_MQMD_* properties have no affect on the encoding of text and numbers written into a JMSMessage.

   A receiving application is likely to assume the values of MQMD.Encoding and MQMD.CodedCharSetId correspond to the encoding and character set of numbers and text in the message body. If JMS_IBM_MQMD_* properties are used, it is the responsibility of the sending application to make it so. The encoding and character set of numbers and text in the message body are set by the JMS_IBM_* properties.

   The badly coded snippet in Figure 39 on page 242 sends a message encoded in character set 1208, with MQMD.CodedCharSetId set to 37.

   a. Send wrongly encoded message

   ```
   TextMessage tmo = session.createTextMessage();
   ((MQDestination) destination).setMessageBodyStyle
                   (WMQConstants.WMQ_MESSAGE_BODY_MQ);
   ((MQDestination)destination).setMQMDWriteEnabled(true);
   tmo.setIntProperty(WMQConstants.JMS_IBM_MQMD_CODEDCHARSETID, 37);
   tmo.setIntProperty(WMQConstants.JMS_IBM_CHARACTER_SET, 1208);
   tmo.setText("String one");
   producer.send(tmo);
   ```

   b. Receiving the message, relying on the value of JMS_IBM_CHARACTER_SET set by the value of MQMD.CodedCharSetId:

   ```
   TextMessage tmi = (TextMessage) cons.receive();
   System.out.println("Message is \"" + tmi.getText() + "\"");
   ```

   c. Resulting output:

   ```
   Message is "éÈÊ'>...??>?"
   ```

Figure 39. Inconsistently coded MQMD and message data

Either of the snippets of code in Figure 40 on page 243 results in a message being put to a queue or topic, with its body containing the application payload without an automatically generated MQRFH2 header being added.

1. Setting WMQ_MESSAGE_BODY_MQ:

```
((MQDestination) destination).setMessageBodyStyle
                (WMQConstants.WMQ_MESSAGE_BODY_MQ);
```

2. Setting WMQ_TARGET_DEST_MQ:

```
((MQDestination) destination).setMessageBodyStyle
                (WMQConstants.WMQ_MESSAGE_BODY_UNSPECIFIED);
((MQDestination) destination).
                setTargetClient(WMQConstants.WMQ_TARGET_DEST_MQ);
```

Figure 40. Send a message with an MQ message body.

Receiving a message

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_JMS, the inbound JMS message type and body are determined by the contents of the received WebSphere MQ message. The message type and body are determined by fields in the MQRFH2 header, or in the MQMD, if there is no MQRFH2.

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_MQ, the inbound JMS message type is JMSBytesMessage. The JMS message body is the message data returned by the underlying MQGET API call. The length of message body is the length returned by the MQGET call. The character set and encoding of the data in the message body is determined by the CodedCharSetId and Encoding fields of the MQMD. The format of the data in the message body is determined by the Format field of the MQMD

If WMQ_MESSAGE_BODY is set to WMQ_MESSAGE_BODY_UNSPECIFIED, the default value, IBM MQ classes for JMS sets it to WMQ_MESSAGE_BODY_JMS.

When you receive a JMSBytesMessage, you can decode it by reference to the following properties:

- JMS_IBM_Format or JMS_IBM_MQMD_Format: This property specifies the format of the IBM MQ header or application payload that starts the JMS message body if there is no preceding WebSphere MQ header.
- JMS_IBM_Character_Set or JMS_IBM_MQMD_CodedCharSetId: This property specifies the CCSID of the IBM MQ header or application payload that starts the JMS message body if there is no preceding WebSphere MQ header.
- JMS_IBM_Encoding or JMS_IBM_MQMD_Encoding: This property specifies the encoding of the IBM MQ header or application payload that starts the JMS message body if there is no preceding WebSphere MQ header.

The following code snippet results in a received message that is a JMSBytesMessage. Irrespective of the content of the received message and of the format field of the received MQMD, the message is a JMSBytesMessage.

```
((MQDestination)destination).setMessageBodyStyle
                            (WMQConstants.WMQ_MESSAGE_BODY_MQ);
```

Destination property WMQ_MESSAGE_BODY
WMQ_MESSAGE_BODY determines whether a JMS application processes the MQRFH2 of an IBM MQ message as part of the message payload (that is, as part of the JMS message body).

Table 39. Property names and descriptions

Property	Short form	Description
WMQ_MESSAGE_BODY	MBODY	Whether a JMS application processes the MQRFH2 of an IBM MQ message as part of the message payload (that is, as part of the JMS message body).

Table 40. Property names, values, and set methods

Property	Valid values in administration tool (defaults in bold)	Valid values in programs	Set method
WMQ_MESSAGE_BODY	• **UNSPECIFIED** When sending, IBM MQ classes for JMS does or does not generate and include an MQRFH2 header, depending on the value of WMQ_TARGET_CLIENT. When receiving, acts as value JMS. • JMS When sending, IBM MQ classes for JMS automatically generates an MQRFH2 header and includes it in the IBM MQ message. When receiving, IBM MQ classes for JMS set the JMS message properties according to values in the MQRFH2 (if present); it does not present the MQRFH2 as part of the JMS message body. • MQ When sending, IBM MQ classes for JMS does not generate an MQRFH2. When receiving, IBM MQ classes for JMS presents the MQRFH2 as part of the JMS message body.	• **WMQ_MESSAGE_BODY_UNSPECIFIED** • WMQ_MESSAGE_BODY_JMS • WMQ_MESSAGE_BODY_MQ	setMessageBodyStyle

JMS persistent messages

IBM MQ classes for JMS applications can use the **NonPersistentMessageClass** queue attribute to provide better performance for JMS persistent messages, at the expense of some reliability.

An IBM MQ queue has an attribute called **NonPersistentMessageClass**. The value of this attribute determines whether nonpersistent messages on the queue are discarded when the queue manager restarts.

You can set the attribute for a local queue by using the IBM MQ Script (MQSC) command, DEFINE QLOCAL, with either of the following parameters:

NPMCLASS(NORMAL)
Nonpersistent messages on the queue are discarded when the queue manager restarts. This is the default value.

NPMCLASS(HIGH)
Nonpersistent messages on the queue are not discarded when the queue manager restarts following a quiesced or immediate shutdown. Nonpersistent messages might be discarded, however, following a preemptive shutdown or a failure.

This topic describes how IBM MQ classes for JMS applications can use this queue attribute to provide better performance for JMS persistent messages.

The PERSISTENCE property of a Queue or Topic object can have the value HIGH. You can use the IBM MQ JMS administration tool to set this value, or an application can call the Destination.setPersistence() method passing the value WMQConstants.WMQ_PER_NPHIGH as a parameter.

If an application sends a JMS persistent message or a JMS nonpersistent message to a destination where the PERSISTENCE property has the value HIGH, and the underlying IBM MQ queue is set to NPMCLASS(HIGH), the message is put on the queue as an IBM MQ nonpersistent message. If the PERSISTENCE property of the destination does not have the value HIGH, or if the underlying queue is set to NPMCLASS(NORMAL), a JMS persistent message is put on the queue as an IBM MQ persistent message, and a JMS nonpersistent message is put on the queue as an IBM MQ nonpersistent message.

If a JMS persistent message is put on a queue as an IBM MQ nonpersistent message, and you want to ensure that the message is not discarded following a quiesced or immediate shutdown of a queue manager, all queues through which the message might be routed must be set to NPMCLASS(HIGH). In the publish/subscribe domain, these queues include subscriber queues. As an aid to enforcing this configuration, IBM MQ classes for JMS throws an InvalidDestinationException if an application tries to create a message consumer for a destination where the PERSISTENCE property has the value HIGH and the underlying IBM MQ queue is set to NPMCLASS(NORMAL).

Setting the PERSISTENCE property of a destination to HIGH does not affect how a message is received from that destination. A message sent as a JMS persistent message is received as a JMS persistent message, and a message sent as a JMS nonpersistent message is received as a JMS nonpersistent message.

When an application sends the first message to a destination where the PERSISTENCE property has the value HIGH, or when an application creates the first message consumer for a destination where the PERSISTENCE property has the value HIGH, IBM MQ classes for JMS issues an MQINQ call to determine whether NPMCLASS(HIGH) is set on the underlying IBM MQ queue. The application must therefore have the authority to inquire on the queue. In addition, IBM MQ classes for JMS preserves the result of the MQINQ call until the destination is deleted, and does not issue more MQINQ calls. Therefore, if you change the NPMCLASS setting on the underlying queue while the application is still using the destination, IBM MQ classes for JMS does not notice the new setting.

By allowing JMS persistent messages to be put on IBM MQ queues as IBM MQ nonpersistent messages, you are gaining performance at the expense of some reliability. If you require maximum reliability for JMS persistent messages, do not send the messages to a destination where the PERSISTENCE property has the value HIGH.

The JMS Layer can use SYSTEM.JMS.TEMPQ.MODEL, instead of SYSTEM.DEFAULT.MODEL.QUEUE. SYSTEM.JMS.TEMPQ.MODEL creates permanent dynamic queues that accept persistent messages, because SYSTEM.DEFAULT.MODEL.QUEUE cannot accept persistent messages. To use temporary queues

to accept persistent messages, you must therefore use SYSTEM.JMS.TEMPQ.MODEL, or change the model queue to an alternative queue of your choosing.

Using TLS with IBM MQ classes for JMS
IBM MQ classes for JMS applications can use Transport Layer Security (TLS) encryption. To do this they require a JSSE provider.

IBM MQ classes for JMS connections using TRANSPORT(CLIENT) support TLS encryption. TLS provides communication encryption, authentication, and message integrity. It is typically used to secure communications between any two peers on the Internet or within an intranet.

IBM MQ classes for JMS uses Java Secure Socket Extension (JSSE) to handle TLS encryption, and therefore requires a JSSE provider. JSE v1.4 JVMs have a JSSE provider built-in. Details of how to manage and store certificates can vary from provider to provider. For information about this, see your JSSE provider's documentation.

This section assumes that your JSSE provider is correctly installed and configured, and that suitable certificates have been installed and made available to your JSSE provider. You can now use JMSAdmin to set a number of administrative properties.

If your IBM MQ classes for JMS application uses a client channel definition table (CCDT) to connect to a queue manager, see “Using a client channel definition table with IBM MQ classes for JMS” on page 273.

SSLCIPHERSUITE object property
Set SSLCIPHERSUITE to enable TLS encryption on a ConnectionFactory object.

To enable TLS encryption on a ConnectionFactory object, use JMSAdmin to set the SSLCIPHERSUITE property to a CipherSuite supported by your JSSE provider. This must match the CipherSpec set on the target channel. However, CipherSuites are distinct from CipherSpecs and therefore have different names. “TLS CipherSpecs and CipherSuites in IBM MQ classes for JMS ” on page 249 contains a table mapping the CipherSpecs supported by IBM MQ to their equivalent CipherSuites as known to JSSE. For more information about CipherSpecs and CipherSuites with IBM MQ, see Securing IBM MQ.

For example, to set up a ConnectionFactory object that can be used to create a connection over an TLS enabled MQI channel with a CipherSpec of TLS_RSA_WITH_AES_128_CBC_SHA, issue the following command to JMSAdmin:

```
ALTER CF(my.cf) SSLCIPHERSUITE(SSL_RSA_WITH_AES_128_CBC_SHA)
```

This can also be set from an application, using the setSSLCipherSuite() method on an MQConnectionFactory object.

For convenience, if a CipherSpec is specified on the SSLCIPHERSUITE property, JMSAdmin attempts to map the CipherSpec to an appropriate CipherSuite and issues a warning. This attempt to map is not made if the property is specified by an application.

Alternatively, use the Client Channel Definition Table (CCDT). For more information, see “Using a client channel definition table with IBM MQ classes for JMS” on page 273.

SSLFIPSREQUIRED object property
If you require a connection to use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS), set the SSLFIPSREQUIRED property of the connection factory to YES.

Note: On AIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status. Customers should view the IBM Crypto for C (ICC) certificate and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the NIST CMVP modules in process list.

The IBM MQ Operator 3.2.0 and queue manager container image 9.4.0.0 onwards are based on UBI 9. FIPS 140-3 compliance is currently pending and its status can be viewed by searching for "Red Hat Enterprise Linux 9 - OpenSSL FIPS Provider" in the NIST CMVP modules in process list.

The default value of this property is NO, which means that a connection can use any CipherSuite that is supported by IBM MQ.

If an application uses more than one connection, the value of SSLFIPSREQUIRED that is used when the application creates the first connection determines the value that is used when the application creates any subsequent connection. This means that the value of the SSLFIPSREQUIRED property of the connection factory that is used to create a subsequent connection is ignored. You must restart the application if you want to use a different value of SSLFIPSREQUIRED.

An application can set this property by calling the setSSLFipsRequired() method of a ConnectionFactory object. The property is ignored if no CipherSuite is set.

Related tasks

Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client

Related reference

Federal Information Processing Standards (FIPS) for AIX, Linux, and Windows

SSLPEERNAME object property

Use SSLPEERNAME to specify a distinguished name pattern, to ensure that your JMS application connects to the correct queue manager.

A JMS application can ensure that it connects to the correct queue manager by specifying a distinguished name (DN) pattern. The connection succeeds only if the queue manager presents a DN that matches the pattern. For more details of the format of this pattern, see the related topics.

The DN is set using the SSLPEERNAME property of a ConnectionFactory object. For example, the following JMSAdmin command sets a ConnectionFactory object to expect the queue manager to identify itself with a Common Name beginning with the characters `QMGR.`, and with at least two Organizational Unit names, the first of which must be IBM and the second WEBSPHERE:

```
ALTER CF(my.cf) SSLPEERNAME(CN=QMGR.*, OU=IBM, OU=WEBSPHERE)
```

Checking is not case sensitive and semicolons can be used in place of commas. SSLPEERNAME can also be set from an application using the setSSLPeerName() method on an MQConnectionFactory object. If this property is not set, no checking is performed on the Distinguished Name supplied by the queue manager. This property is ignored if no CipherSuite is set.

SSLCERTSTORES object property

Use SSLCERTSTORES to specify a list of LDAP servers to use for certificate revocation list (CRL) checking.

It is common to use a certificate revocation list (CRL) to identify certificates that are no longer trusted. CRLs are typically hosted on LDAP servers. JMS allows an LDAP server to be specified for CRL checking under Java 2 v1.4 or later. The following JMSAdmin example directs JMS to use a CRL hosted on an LDAP server named crl1.ibm.com:

```
ALTER CF(my.cf) SSLCRL(ldap://crl1.ibm.com)
```

Note: To use a CertStore successfully with a CRL hosted on an LDAP server, make sure that your Java Software Development Kit (SDK) is compatible with the CRL. Some SDKs require that the CRL conforms to RFC 2587, which defines a schema for LDAP v2. Most LDAP v3 servers use RFC 2256 instead.

If your LDAP server is not running on the default port of 389, you can specify the port by appending a colon (:) and the port number to the host name. If the certificate presented by the queue manager is present in the CRL hosted on crl1.ibm.com, the connection is not completed. To avoid a single point of failure, JMS allows multiple LDAP servers to be supplied by supplying a list of LDAP servers delimited by the space character. Here is an example:

```
ALTER CF(my.cf) SSLCRL(ldap://crl1.ibm.com ldap://crl2.ibm.com)
```

When multiple LDAP servers are specified, JMS tries each one in turn until it finds a server with which it can successfully verify the queue manager's certificate. Each server must contain identical information.

A string in this format can be supplied by an application on the MQConnectionFactory.setSSLCertStores() method. Alternatively, the application can create one or more java.security.cert.CertStore objects, place these in a suitable Collection object, and supply this Collection object to the setSSLCertStores() method. In this way, the application can customize CRL checking. See your JSSE documentation for details on constructing and using CertStore objects.

The certificate presented by the queue manager when a connection is being set up is validated as follows:

1. The first CertStore object in the Collection identified by sslCertStores is used to identify a CRL server.
2. An attempt is made to contact the CRL server.
3. If the attempt is successful, the server is searched for a match for the certificate.
   a. If the certificate is found to be revoked, the search process is over and the connection request fails with reason code MQRC_SSL_CERTIFICATE_REVOKED.
   b. If the certificate is not found, the search process is over and the connection is allowed to proceed.
4. If the attempt to contact the server is unsuccessful, the next CertStore object is used to identify a CRL server and the process repeats from step 2.

   If this was the last CertStore in the Collection, or if the Collection contains no CertStore objects, the search process has failed and the connection request fails with reason code MQRC_SSL_CERT_STORE_ERROR.

The Collection object determines the order in which CertStores are used.

If your application uses setSSLCertStores() to set a Collection of CertStore objects, the MQConnectionFactory can no longer be bound into a JNDI namespace. Attempting to do so causes an exception. If the sslCertStores property is not set, no revocation checking is performed on the certificate provided by the queue manager. This property is ignored if no CipherSuite is set.

SSLRESETCOUNT object property
This property represents the total number of bytes sent and received by a connection before the secret key that is used for encryption is renegotiated.

The number of bytes sent is the number before encryption, and the number of bytes received is the number after decryption. The number of bytes also includes control information sent and received by IBM MQ classes for JMS.

For example, to configure a ConnectionFactory object that can be used to create a connection over an TLS enabled MQI channel with a secret key that is renegotiated after 4 MB of data have flowed, issue the following command to JMSAdmin:

```
ALTER CF(my.cf) SSLRESETCOUNT(4194304)
```

An application can set this property by calling the setSSLResetCount() method of a ConnectionFactory object.

If the value of this property is zero, which is the default value, the secret key is never renegotiated. The property is ignored if no CipherSuite is set.

SSLSocketFactory object property
To customize other aspects of the TLS connection for an application, create an SSLSocketFactory and configure JMS to use it.

You might want to customize other aspects of the TLS connection for an application. For example, you might want to initialize cryptographic hardware or change the keystore and truststore in use. To do this, the application must first create a javax.net.ssl.SSLSocketFactory object that is customized accordingly. See your JSSE documentation for information about how to do this, because the customizable features vary from provider to provider. After a suitable SSLSocketFactory object is obtained, use

the MQConnectionFactory.setSSLSocketFactory() method to configure JMS to use the customized SSLSocketFactory object.

If your application uses the setSSLSocketFactory() method to set a customized SSLSocketFactory object, the MQConnectionFactory object can no longer be bound into a JNDI namespace. Attempting to do so causes an exception. If this property is not set, the default SSLSocketFactory object is used. See your JSSE documentation for details of the behavior of the default SSLSocketFactory object. This property is ignored if no CipherSuite is set.

Important: Do not assume that the use of the SSL properties ensures security when a ConnectionFactory object is retrieved from a JNDI namespace that is not itself secure. Specifically, the standard LDAP implementation of JNDI is not secure. An attacker can imitate the LDAP server, misleading a JMS application into connecting to the wrong server without noticing. With suitable security arrangements in place, other implementations of JNDI (such as the fscontext implementation) are secure.

Making changes to the JSSE keystore or truststore
If you make changes to the keystore or truststore, you must take certain actions for the changes to be picked up.

If you change the contents of the JSSE keystore or truststore, or change the location of the keystore or truststore file, IBM MQ classes for JMS applications that are running at the time do not automatically pick up the changes. For the changes to take effect, the following actions must be performed:

- The applications must close all their connections, and destroy any unused connections in connection pools.
- If your JSSE provider caches information from the keystore and truststore, this information must be refreshed.

After these actions have been performed, the applications can then re-create their connections.

Depending on how you design your applications, and on the function provided by your JSSE provider, it might be possible to perform these actions without stopping and restarting your applications. However, stopping and restarting the applications might be the simplest solution.

TLS CipherSpecs and CipherSuites in IBM MQ classes for JMS
The ability of IBM MQ classes for JMS applications to establish connections to a queue manager, depends on the CipherSpec specified at the server end of the MQI channel and the CipherSuite specified at the client end.

The following table lists the CipherSpecs supported by IBM MQ and their equivalent CipherSuites.

Deprecated You should review the topic Deprecated CipherSpecs to see if any of the CipherSpecs, listed in the following table, have been deprecated by IBM MQ and, if so, at which update the CipherSpec was deprecated.

Important: The CipherSuites listed are those supported by the IBM Java Runtime Environment (JRE) supplied with IBM MQ. The CipherSuites that are listed include those supported by the Oracle Java JRE. For more information about configuring your application to use an Oracle Java JRE, see Configuring your application to use IBM Java or Oracle Java CipherSuite mappings.

The table also indicates the protocol that is used for the communication, and whether or not the CipherSuite conforms to the FIPS 140-2 standard.

Note: On AIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status. Customers should view the IBM Crypto for C (ICC) certificate and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the NIST CMVP modules in process list.

The IBM MQ Operator 3.2.0 and queue manager container image 9.4.0.0 onwards are based on UBI 9. FIPS 140-3 compliance is currently pending and its status can be viewed by searching for "Red Hat Enterprise Linux 9 - OpenSSL FIPS Provider" in the NIST CMVP modules in process list.

Ciphersuites denoted as FIPS 140-2 compliant can be used if the application has not been configured to enforce FIPS 140-2 compliance, but if FIPS 140-2 compliance has been configured for the application (see the following notes on configuration) only those CipherSuites which are marked as FIPS 140-2 compatible can be configured; attempting to use other CipherSuites results in an error.

Note: Each JRE can have multiple cryptographic security providers, each of which can contribute an implementation of the same CipherSuite. However, not all security providers are FIPS 140-2 certified. If FIPS 140-2 compliance is not enforced for an application then it is possible that an uncertified implementation of the CipherSuite might be used. Uncertified implementations might not operate in compliance with FIPS 140-2, even if the CipherSuite theoretically meets the minimum security level required by the standard. See the following notes for more information about configuring FIPS 140-2 enforcement in IBM MQ JMS applications.

For more information about FIPS 140-2 and Suite-B compliance for CipherSpecs and CipherSuites, see Specifying CipherSpecs. You might also need to be aware of information that concerns US Federal Information Processing Standards.

To use the full set of CipherSuites and to operate with certified FIPS 140-2 and/or Suite-B compliance, a suitable JRE is required. IBM Java 7 Service Refresh 4 Fix Pack 2 or a higher level of IBM JRE provides the appropriate support for the TLS 1.2 CipherSuites listed in Table 41 on page 251.

To be able to use TLS 1.3 Ciphers, the JRE running your application must support TLS 1.3.

Note: To use some CipherSuites, the 'unrestricted' policy files need to be configured in the JRE. For more details of how policy files are set up in an SDK or JRE, see the *IBM SDK Policy files* topic in the *Security Reference for IBM SDK, Java Technology Edition* for the version you are using.

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_3DES_EDE_CBC_SHA256	SSL_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_128_CBC_SHA256	SSL_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_128_GCM_SHA256	SSL_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 269	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_256_CBC_SHA384	SSL_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_256_GCM_SHA384	SSL_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS 1.2	yes
ECDHE_ECDSA_NULL_SHA256	SSL_ECDHE_ECDSA_WITH_NULL_SHA	TLS_ECDHE_ECDSA_WITH_NULL_SHA	TLS 1.2	no

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 269	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_RC4_128_SHA256	SSL_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS 1.2	no
ECDHE_RSA_3DES_EDE_CBC_SHA256	SSL_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_128_CBC_SHA256	SSL_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 269	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_128_GCM_SHA256	SSL_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_256_CBC_SHA384	SSL_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS 1.2	yes

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_256_GCM_SHA384	SSL_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2	yes
ECDHE_RSA_NULL_SHA256	SSL_ECDHE_RSA_WITH_NULL_SHA	TLS_ECDHE_RSA_WITH_NULL_SHA	TLS 1.2	no

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_RC4_128_SHA256	SSL_ECDHE_RSA_WITH_RC4_128_SHA	TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLS 1.2	no
TLS_RSA_WITH_3DES_EDE_CBC_SHA "2" on page 269	SSL_RSA_WITH_3DES_EDE_CBC_SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS 1.0	no "4" on page 269

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_AES_128_CBC_SHA	SSL_RSA_WITH_AES_128_CBC_SHA	TLS_RSA_WITH_AES_128_CBC_SHA	TLS 1.0	no "4" on page 269
TLS_RSA_WITH_AES_128_CBC_SHA256	SSL_RSA_WITH_AES_128_CBC_SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	no "4" on page 269

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_AES_128_GCM_SHA256	SSL_RSA_WITH_AES_128_GCM_SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2	no "4" on page 269
TLS_RSA_WITH_AES_256_CBC_SHA	SSL_RSA_WITH_AES_256_CBC_SHA	TLS_RSA_WITH_AES_256_CBC_SHA	TLS 1.0	no "4" on page 269

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_AES_256_CBC_SHA256	SSL_RSA_WITH_AES_256_CBC_SHA256	TLS_RSA_WITH_AES_256_CBC_SHA256	TLS 1.2	no "4" on page 269
TLS_RSA_WITH_AES_256_GCM_SHA384	SSL_RSA_WITH_AES_256_GCM_SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2	no "4" on page 269

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	TLS 1.0	no
TLS_RSA_WITH_NULL_SHA256	SSL_RSA_WITH_NULL_SHA256	TLS_RSA_WITH_NULL_SHA256	TLS 1.2	no
TLS_RSA_WITH_RC4_128_SHA256	SSL_RSA_WITH_RC4_128_SHA	SSL_RSA_WITH_RC4_128_SHA	TLS 1.2	no

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ANY_TLS12	*TLS12	*TLS12	TLS 1.2	yes
TLS_AES_128_GCM_SHA256 "3" on page 269	TLS_AES_128_GCM_SHA256	TLS_AES_128_GCM_SHA256	TLS V1.3	no
TLS_AES_256_GCM_SHA384 "3" on page 269	TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384	TLS V1.3	no

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_CHACHA20_POLY1305_SHA256 "3" on page 269	TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256	TLS V1.3	no
TLS_AES_128_CCM_SHA256 "3" on page 269	TLS_AES_128_CCM_SHA256	TLS_AES_128_CCM_SHA256	TLS V1.3	no

Table 41. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 269**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_AES_128_CCM_8_SHA256 "3" on page 269	TLS_AES_128_CCM_8_SHA256	TLS_AES_128_CCM_8_SHA256	TLS V1.3	no
ANY "3" on page 269	*ANY	*ANY	Multiple	no
ANY_TLS13 "3" on page 269	*TLS13	*TLS13	TLS V13	no
ANY_TLS12_OR_HIGHER "3" on page 269	*TLS12ORHIGHER	*TLS12ORHIGHER	TLS 1.2 and above	no
ANY_TLS13_OR_HIGHER "3" on page 269	*TLS13ORHIGHER	*TLS13ORHIGHER	TLS 1.3 and above	no

Notes:

1. This is the value configured on a channel in IBM MQ, including in a CCDT (binary or JSON).
2. Deprecated CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.
3. To be able to use TLS v1.3 Ciphers, the Java runtime environment (JRE) running your application must support TLS v1.3.
4. V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, the IBM Java 8 JRE removes support for RSA key exchange when operating in FIPS mode.

Configuring Ciphersuites and FIPS-compliance in an IBM MQ classes for JMS application

- An application that uses IBM MQ classes for JMS can use either of two methods to set the CipherSuite for a connection:
  - Call the setSSLCipherSuite method of a ConnectionFactory object.
  - Use the IBM MQ JMS administration tool to set the SSLCIPHERSUITE property of a ConnectionFactory object.
- An application that uses IBM MQ classes for JMS can use either of two methods to enforce FIPS 140-2 compliance:
  - Call the setSSLFipsRequired method of a ConnectionFactory object.
  - Use the IBM MQ JMS administration tool to set the SSLFIPSREQUIRED property of a ConnectionFactory object.

Configuring your application to use IBM Java or Oracle Java CipherSuite mappings

V 9.4.0 From IBM MQ 9.4.0, a Cipher can be defined as either the CipherSpec or CipherSuite name and is handled correctly by IBM MQ.

Note: Removed The Java System Property `com.ibm.mq.cfg.useIBMCipherMappings`, which controlled which mappings were used in earlier versions of IBM MQ, is no longer needed and is removed from the product at IBM MQ 9.4.0.

Interoperability limitations

Certain CipherSuites might be compatible with more than one IBM MQ CipherSpec, depending on the protocol in use. However, only the CipherSuite/CipherSpec combination that uses the TLS version specified in Table 1 is supported. Attempting to use the unsupported combinations of CipherSuites and CipherSpecs will fail with an appropriate exception. Installations using any of these CipherSuite/CipherSpec combinations should move to a supported combination.

The following table shows the CipherSuites to which this limitation applies.

Table 42. CipherSuites and their supported and unsupported CipherSpecs

CipherSuite	Supported TLS CipherSpec	Unsupported SSL CipherSpec
SSL_RSA_WITH_3DES_EDE_CBC_SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA "1" on page 270	TRIPLE_DES_SHA_US
SSL_RSA_WITH_DES_CBC_SHA	TLS_RSA_WITH_DES_CBC_SHA	DES_SHA_EXPORT
SSL_RSA_WITH_RC4_128_SHA	TLS_RSA_WITH_RC4_128_SHA256	RC4_SHA_US

Note:

1. Deprecated This CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.

Writing channel exits in Java for IBM MQ classes for JMS
You create channel exits by defining Java classes that implement specified interfaces.

For an introduction to security exits, start with the Channel security exit programs topic.

Three interfaces are defined in the com.ibm.mq.exits package:

- WMQSendExit, for a send exit
- WMQReceiveExit, for a receive exit
- WMQSecurityExit, for a security exit

The following sample code defines a class that implements all three interfaces:

```
public class MyMQExits implements
WMQSendExit, WMQReceiveExit, WMQSecurityExit {
    // Default constructor
  public MyMQExits(){
  }
    // This method implements the send exit interface
  public ByteBuffer channelSendExit(
                                    MQCXP channelExitParms,
                                    MQCD channelDefinition,
                                    ByteBuffer agentBuffer)
  {
    // Complete the body of the send exit here
  }
    // This method implements the receive exit interface
  public ByteBuffer channelReceiveExit(
                                    MQCXP channelExitParms,
                                    MQCD channelDefinition,
                                    ByteBuffer agentBuffer)
  {
    // Complete the body of the receive exit here
  }
    // This method implements the security exit interface
  public ByteBuffer channelSecurityExit(
                                    MQCXP channelExitParms,
                                    MQCD channelDefinition,
                                    ByteBuffer agentBuffer)
  {
    // Complete the body of the security exit here
  }
}
```

Each exit receives as parameters an MQCXP object and an MQCD object. These objects represent the MQCXP and MQCD structures defined in the procedural interface.

When a send exit is called, the agentBuffer parameter contains the data that is about to be sent to the server queue manager. A length parameter is not required because the expression agentBuffer.limit() provides the length of the data. The send exit returns as its value the data to be sent to the server queue manager. However, if the send exit is not the last send exit in a sequence of send exits, the data returned is passed instead to the next send exit in the sequence. A send exit can return a modified version of the data that it receives in the agentBuffer parameter, or it can return the data unchanged. The simplest possible exit body is therefore:

```
{ return agentBuffer; }
```

When a receive exit is called, the agentBuffer parameter contains the data that has been received from the server queue manager. The receive exit returns as its value the data to be passed to the application by

IBM MQ classes for JMS. However, if the receive exit is not the last receive exit in a sequence of receive exits, the data returned is passed instead to the next receive exit in the sequence.

When a security exit is called, the agentBuffer parameter contains the data that has been received in a security flow from the security exit at the server end of the connection. The security exit returns as its value the data to be sent in a security flow to the server security exit.

Channel exits are called with a buffer that has a backing array. For best performance, the exit should return a buffer with a backing array.

Up to 32 characters of user data can be passed to a channel exit when it is called. The exit accesses the user data by calling the getExitData() method of the MQCXP object. Although the exit can change the user data by calling the setExitData() method, the user data is refreshed every time the exit is called. Any changes made to the user data are therefore lost. However, the exit can pass data from one call to the next by using the exit user area of the MQCXP object. The exit accesses the exit user area by reference by calling the getExitUserArea() method.

Every exit class must have a constructor. The constructor can be either the default constructor, as shown in the previous example, or a constructor with a string parameter. The constructor is called to create an instance of the exit class for each exit defined in the class. Therefore, in the previous example, an instance of the MyMQExits class is created for the send exit, another instance is created for the receive exit, and a third instance is created for the security exit. When a constructor with a string parameter is called, the parameter contains the same user data that is passed to the channel exit for which the instance is being created. If an exit class has both a default constructor and a single parameter constructor, the single parameter constructor takes precedence.

Do not close the connection from within a channel exit.

When data is sent to the server end of a connection, TLS encryption is performed *after* any channel exits are called. Similarly, when data is received from the server end of a connection, TLS decryption is performed *before* any channel exits are called.

In versions of IBM MQ classes for JMS earlier than IBM WebSphere MQ 7.0, channel exits were implemented using the interfaces MQSendExit, MQReceiveExit, and MQSecurityExit. You can still use these interfaces, but the new interfaces are preferred for improved function and performance.

Configuring IBM MQ classes for JMS to use channel exits
An IBM MQ classes for JMS application can use channel security, send, and receive exits on the MQI channel that starts when the application connects to a queue manager. The application can use exits written in Java, C, or C++. The application can also use a sequence of send or receive exits that are run in succession.

The following properties are used specify a send exit, or a sequence of send exits, used by a JMS connection:

- The **SENDEXIT** property of an MQConnectionFactory object.
- The **sendexit** property on an activation specification used by the IBM MQ resource adapter for inbound communication,
- The **sendexit** property on a ConnectionFactory object used by the IBM MQ resource adapter for output communication.

The value of the property is a string that comprises one or more items separated by commas. Each item identifies a send exit in one of the following ways:

- The name of a class that implements the `WMQSendExit` interface for a send exit written in Java.
- A string in the format *libraryName* (*entryPointName*) for a send exit written in C or C++.

In a similar way, the following properties specify the receive exit, or sequence of receive exits, used by a connection:

- The **RECEXIT** property of an MQConnectionFactory object.
- The **receiveexit** property on an activation specification used by the IBM MQ resource adapter for inbound communication,

- The **receiveexit** property on a ConnectionFactory object used by the IBM MQ resource adapter for output communication.

The following properties specify the security exit used by a connection:

- The **SECEXIT** property of an MQConnectionFactory object.
- The **securityexit** property on an activation specification used by the IBM MQ resource adapter for inbound communication,
- The **securityexit** property on a ConnectionFactory object used by the IBM MQ resource adapter for output communication.

For MQConnectionFactories, you can set the **SENDEXIT**, **RECEXIT** and **SECEXIT** properties by using the IBM MQ JMS administration tool or IBM MQ Explorer. Alternatively, an application can set the properties by calling the `setSendExit()`, `setReceiveExit()`, and `setSecurityExit()` methods.

Channel exits are loaded by their own class loader. To find a channel exit, the class loader searches the following locations in the specified order.

1. The class path specified by the property **com.ibm.mq.cfg.ClientExitPath.JavaExitsClasspath** or by the **JavaExitsClassPath** attribute in the Channels stanza of the IBM MQ client configuration file.
2. Deprecated The class path specified by the Java system property **com.ibm.mq.exitClasspath**. Note that this property is now deprecated.
3. The IBM MQ exits directory, as shown in Table 43 on page 272. The class loader first searches the directory for class files that are not packaged in Java archive (JAR) files. If the channel exit is not found, the class loader then searches the JAR files in the directory.

Table 43. The IBM MQ exits directory

Platform	Directory
Linux AIX AIX and Linux	`/var/mqm/exits` (32-bit channel exits) `/var/mqm/exits64` (64-bit channel exits)
Windows Windows	*install_data_dir*\exits where *install_data_dir* is the directory that you chose for the IBM MQ data files during installation. The default directory is C:\ProgramData\IBM\MQ.

Note: If a channel exit exists in more than one location, the IBM MQ classes for JMS loads the first instance that it finds.

The parent of the class loader is the class loader that is used to load IBM MQ classes for JMS. It is therefore possible for the parent class loader to load a channel exit if it cannot be found in any of the preceding locations. However, when you are using the IBM MQ classes for JMS in an environment such as a JEE application server, you are not likely to be able to influence the choice of the parent class loader and so the class loader should be configured by setting the Java system property **com.ibm.mq.cfg.ClientExitPath.JavaExitsClasspath** on the application server.

If your application is being run with the Java security manager enabled, then the policy configuration file used by the Java runtime environment that the application is running in must have the permissions to load a channel exit class. For information on how to do this, see Running IBM MQ classes for JMS applications under the Java Security Manager.

The `MQSendExit`, `MQReceiveExit`, and `MQSecurityExit` interfaces supplied with versions earlier than IBM WebSphere MQ 7.0 are still supported. If you use channel exits that implement these interfaces, `com.ibm.mq.jar` must be present in the class path.

For information about how to write channel exits in C, see “Channel-exit programs for messaging channels” on page 926. You must store channel exit programs written in C or C++ in the directory shown in Table 43 on page 272.

If your application uses a client channel definition table (CCDT) to connect to a queue manager, see “Using a client channel definition table with IBM MQ classes for JMS” on page 273.

Specifying the user data to be passed to channel exits when using IBM MQ classes for JMS
Up to 32 characters of user data can be passed to a channel exit when it is called.

The SENDEXITINIT property of an MQConnectionFactory object specifies the user data that is passed to each send exit when it is called. The value of the property is a string that comprises one or more items of user data separated by commas. The position of each item of user data within the string determines which send exit, in a sequence of send exits, the user data is passed to. For example, the first item of user data in the string is passed to the first send exit in a sequence of send exits.

You can set the SENDEXITINIT property by using the IBM MQ JMS administration tool or IBM MQ Explorer. Alternatively, an application can set the property by calling the setSendExitInit() method.

In a similar way, the RECEXITINIT property of a ConnectionFactory object specifies the user data that is passed to each receive exit, and the SECEXITINIT property specifies the user data passed to a security exit. You can set these properties by using the IBM MQ JMS administration tool or IBM MQ Explorer. Alternatively, an application can set the properties by calling the setReceiveExitInit() and setSecurityExitInit() methods.

Note the following rules when specifying user data that is passed to channel exits:

- If the number of items of user data in a string is more than the number of exits in a sequence, the excess items of user data are ignored.
- If the number of items of user data in a string is less than the number of exits in a sequence, each unspecified item of user data is set to an empty string. Two commas in succession within a string, or a comma at the beginning of a string, also denotes an unspecified item of user data.

If an application uses a client channel definition table (CCDT) to connect to a queue manager, any user data specified in a client connection channel definition is passed to channel exits when they are called. For more information about using a client channel definition table, see “Using a client channel definition table with IBM MQ classes for JMS” on page 273.

Using a client channel definition table with IBM MQ classes for JMS
An IBM MQ classes for JMS application can use client connection channel definitions that are stored in a client channel definition table (CCDT). You configure a ConnectionFactory object to use the CCDT. There are some restrictions on its use.

As an alternative to creating a client connection channel definition by setting certain properties of a ConnectionFactory object, an IBM MQ classes for JMS application can use client connection channel definitions that are stored in a client channel definition table. These definitions are created by IBM MQ Script (MQSC) commands or IBM MQ Programmable Command Format (PCF) commands. When the application creates a Connection object, IBM MQ classes for JMS searches the client channel definition table for a suitable client connection channel definition, and uses the channel definition to start an MQI channel. For more information about client channel definition tables and how to construct one, see Client channel definition table.

To use a client channel definition table, the CCDTURL property of a ConnectionFactory object must be set to a URL object. IBM MQ classes for JMS do not read the information about the CCDT from the IBM MQ MQI client configuration file, although some other values are used from there (see “The IBM MQ classes for JMS/Jakarta Messaging configuration file” on page 94 for which value apply). The URL object encapsulates a uniform resource locator (URL) that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed. You can set the CCDTURL property by using the IBM MQ JMS administration tool, or an application can set the property by creating a URL object and calling the setCCDTURL() method of the ConnectionFactory object.

For example, if the file ccdt1.tab contains a client channel definition table and is stored on the same system on which the application is running, the application can set the CCDTURL property in the following way:

```
java.net.URL chanTab1 = new URL("file:///home/admdata/ccdt1.tab");
factory.setCCDTURL(chanTab1);
```

As another example, suppose the file ccdt2.tab contains a client channel definition table and is stored on a system that is different from the one on which the application is running. If the file can be accessed using the FTP protocol, the application can set the CCDTURL property in the following way:

```
java.net.URL chanTab2 = new URL("ftp://ftp.server/admdata/ccdt2.tab");
factory.setCCDTURL(chanTab2);
```

In addition to setting the CCDTURL property of the ConnectionFactory object, the QMANAGER property of the same object must be set to one of the following values:

- The name of a queue manager
- An asterisk (*) followed by the name of a queue manager group

These are the same values that can be used for the **QMgrName** parameter on an MQCONN call issued by a client application that is using Message Queue Interface (MQI). For more information about the meaning of these values therefore, see MQCONN. You can set the QMANAGER property by using the IBM MQ JMS administration tool or IBM MQ Explorer. Alternatively, an application can set the property by calling the setQueueManager() method of the ConnectionFactory object.

If an application then creates a Connection object from the ConnectionFactory object, IBM MQ classes for JMS accesses the client channel definition table identified by the CCDTURL property, uses the QMANAGER property to search the table for a suitable client connection channel definition, and then uses the channel definition to start an MQI channel to a queue manager.

Note that the CCDTURL and CHANNEL properties of a ConnectionFactory object cannot both be set when the application calls the createConnection() method. If both properties are set, the method throws an exception. The CCDTURL or CHANNEL property is considered to be set if its value is anything other than null, an empty string, or a string containing all blank characters.

When IBM MQ classes for JMS finds a suitable client connection channel definition in the client channel definition table, it uses only the information extracted from the table to start an MQI channel. Any channel related properties of the ConnectionFactory object are ignored.

In particular, note the following points if you are using TLS:

- An MQI channel uses TLS only if the channel definition extracted from the client channel definition table specifies the name of a CipherSpec supported by IBM MQ classes for JMS.
- A client channel definition table also contains information about the location of Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs). IBM MQ classes for JMS uses only this information to access LDAP servers that hold CRLs.
- A client channel definition table can also contain the location of an OCSP responder. IBM MQ classes for JMS cannot use the OCSP information in a client channel definition table file. However, you can configure OCSP as described in the section Online Certificate Status Protocol (OCSP) in Java and JMS client applications.

For more information about using TLS with a client channel definition table, see Using the extended transactional client with TLS channels.

Note also the following points if you are using channel exits:

- An MQI channel uses only the channel exits and associated user data specified by the channel definition extracted from the client channel definition table.
- A channel definition extracted from a client channel definition table can specify channel exits that are written in Java. This means, for example, that the SCYEXIT parameter on the DEFINE CHANNEL

command to create a client connection channel definition can specify the name of a class that implements the WMQSecurityExit interface. Similarly, the SENDEXIT parameter can specify the name of a class that implements the WMQSendExit interface, and the RCVEXIT parameter can specify the name of a class that implements the WMQReceiveExit interface. For more information about how to write a channel exit in Java, see"Writing channel exits in Java for IBM MQ classes for JMS" on page 270.

The use of channel exits written in a language other than Java is also supported. For information about how to specify the SCYEXIT, SENDEXIT, and RCVEXIT parameters on the DEFINE CHANNEL command for channel exits written in another language, see DEFINE CHANNEL.

Automatic JMS client reconnection
Configure your JMS client to reconnect automatically following a network, queue manager, or server failure.

Normally, if a stand-alone IBM MQ classes for JMS application is connected to a queue manager by using the client transport, and the queue manager becomes unavailable for some reason (due to a network outage, a queue manager failure, or the queue manager being stopped, for example), the IBM MQ classes for JMS will throw a JMSException the next time the application tries to communicate with the queue manager. The application must catch the JMSException and attempt to reconnect to the queue manager. You can simplify the design of the application by enabling automatic client reconnection. When the queue manager becomes unavailable, the IBM MQ classes for JMS attempts to reconnect to the queue manager automatically on behalf of the application. This means that the application does not need to contain logic to reconnect.

The use of this implementation of automatic client reconnection is not supported within Java Platform, Enterprise Edition application servers. See "Using automatic client reconnection in Java EE environments" on page 281 for an alternative implementation.

Using automatic JMS client reconnection
If a stand-alone IBM MQ classes for JMS application uses a Connection Factory that has the CONNECTIONNAMELIST or the CCDTURL property set, the application is eligible to use automatic client reconnection.

Automatic client reconnection can be used to reconnect to queue managers including those that are part of a high availability (HA) configuration. HA configurations include multi-instance queue managers, RDQM queue managers, or HA queue managers on an IBM MQ appliance.

The behavior of the automatic client reconnection functionality that is provided by the IBM MQ classes for JMS depends on the properties that follow:

The JMS Connection Factory property TRANSPORT (Short name TRAN)

TRANSPORT specifies how applications that use the Connection Factory connects to a queue manager. This property must be set to the value CLIENT for automatic client reconnection to be used. Automatic client reconnection is not available to applications that connect to a queue manager that uses a Connection Factory that has the TRANSPORT property set to BIND, DIRECT, or DIRECTHTTP.

The JMS Connection Factory property QMANAGER (Short name QMGR)

The QMANAGER property specifies the name of the queue manager that the Connection Factory connects to.

The JMS Connection Factory property CONNECTIONNAMELIST (Short name CRHOSTS)

The CONNECTIONNAMELIST property is a comma-separated list, where each entry contains information on the host name and port that are to be used to connect to the queue manager specified by the QMANAGER property when you are using the CLIENT transport. The list has the following format: host name(port), host name(port).

The JMS Connection Factory property CCDTURL (Short name CCDT)

The CCDTURL property points to the client channel definition table that the IBM MQ classes for JMS uses when it connects to a queue manager using a CCDT.

The JMS Connection Factory property CLIENTRECONNECTOPTIONS (Short name CROPT)

CLIENTRECONNECTOPTIONS controls whether the IBM MQ classes for JMS will attempt to automatically connect to a queue manager on behalf of an application if a queue manager becomes available.

The DefRecon attribute in the Channels stanza of the client configuration file

The DefRecon attribute provides an administrative option to enable all applications to automatically reconnect, or to disable the automatic reconnection for applications that are written to reconnect automatically.

Automatic client reconnection is only available when an application successfully connects to a queue manager.

When an application connects to a queue manager that uses the CLIENT transport, the IBM MQ classes for JMS use the value of the Connection Factory property CLIENTRECONNECTOPTIONS to determine whether to use automatic client reconnection, if the queue manager that the application is connected to becomes unavailable. Table 1 shows the possible values for the CLIENTRECONNECTOPTIONS property, and the behavior of the IBM MQ classes for JMS for each of these values:

Table 44. Possible CLIENTRECCECTOPTIONS property values.

CLIENTRECONNECTOPTIONS	**Behavior of IBM MQ classes for JMS**
ANY	If CONNECTIONNAMELIST is set, use the value of the CONNECTIONNAMELIST property to open a connection to a host name and port combination, and connect to any queue manager. In order to use this automatic client reconnection option, the QMANAGER property must be set to either the default value or "*". If CCDTURL is set, open the client channel definition table that is specified by the CCDTURL property, pick an entry in the table, and then use that entry to start a client connection channel to a queue manager. To use this automatic client reconnection option, the QMANAGER property must be set to either: • An asterisk (*) • An asterisk (*) followed by the name of a queue manager group • An empty string, or a string that contains all blank characters
ASDEF	Use the value of DefRecon to determine whether automatic client reconnection is available.
DISABLED	Do not perform any automatic client reconnection and return a JMSException to the application.

Table 44. Possible CLIENTRECCECTOPTIONS property values. (continued)	
CLIENTRECONNECTOPTIONS	**Behavior of IBM MQ classes for JMS**
QMGR	Specifies that the client must reconnect to the same queue manager. This option must be used for high availability solutions, where reconnection to another instance of the same queue manager is required. If CONNECTIONNAMELIST is set, use the value of the CONNECTIONNAMELIST property to open a connection to a host name and port combination, and connect to the queue manager specified by the QMANAGER property. If CCDTURL is set, open the client channel definition table that is specified by the CCDTURL property, find the entries in the table that match the queue manager name that is specified by the QMANAGER property and then use those entries to start a client connection channel to that queue manager.

If CONNECTIONNAMELIST is set, when you perform automatic client reconnection, the IBM MQ classes for JMS uses the information in the Connection Factory property CONNNECTIONNAMELIST to determine what system to reconnect to.

The IBM MQ classes for JMS initially tries to reconnect by using the host name and port that is specified in the first entry in the CONNECTIONNAMELIST. If a connection is made, the IBM MQ classes for JMS then tries to connect to the queue manager that has the name specified in the QMANAGER property. If a connection to the queue manager can be established, the IBM MQ classes for JMS reopens all of the IBM MQ objects that the application had open before automatic client reconnection and continue running as before.

If a connection cannot be established to the required queue manager by using the first entry in the CONNECTIONNAMELIST, the IBM MQ classes for JMS tries the second entry in the CONNECTIONNAMELIST, and so on.

When the IBM MQ classes for JMS have tried all of the entries in the CONNECTIONNAMELIST, they wait for a period of time before they try to reconnect again. To perform the new reconnection attempt, the IBM MQ classes for JMS start with the first entry in the CONNECTIONNAMELIST. They then try each entry in the CONNECTIONNAMELIST in turn until either a reconnection occurs or the end of the CONNECTIONNAMELIST is reached, at which point the IBM MQ classes for JMS waits for a period of time before they try again.

If CCDTURL is set, when performing automatic client reconnection, the IBM MQ classes for JMS uses the client channel definition table that is specified in the CCDTURL property to determine what system to reconnect to.

The IBM MQ classes for JMS initially parses the client channel definition table and finds a suitable entry that matches the value of the QMANAGER property. When an entry is found, the IBM MQ classes for JMS tries to reconnect to the required queue manager using that entry. If a connection to the queue manager can be established, the IBM MQ classes for JMS reopens all of the IBM MQ objects that the application had open before automatic client reconnection and continue running as before.

If a connection cannot be established to the required queue manager, the IBM MQ classes for JMS looks for another suitable entry in the client channel definition table and tries to use that, and so on.

When the IBM MQ classes for JMS have tried all of the suitable entries in the client channel definition table, they wait for a period of time before trying to reconnect again. To perform the new reconnection attempt, the IBM MQ classes for JMS parses the client channel definition table again and tries the first

suitable entry. They will then try each suitable entry in the client channel definition table in turn until either a reconnection occurs or the last suitable entry in the client channel definition table has been tried, at which point the IBM MQ classes for JMS waits for a period of time before trying again.

Whether using CONNECTIONNAMELIST, or CCDTURL, the process of automatic client reconnection continues until the IBM MQ classes for JMS successfully reconnected to the queue manager specified by the QMANAGER property.

By default, the reconnection attempts happen at the following intervals:

- The first attempt is made after an initial delay of 1 second, plus a random element up to 250 milliseconds.
- The second attempt is made 2 seconds, plus a random interval of up to 500 milliseconds, after the first attempt fails.
- The third attempt is made 4 seconds, plus a random interval of up to 1 second, after the second attempt fails.
- The fourth attempt is made 8 seconds, plus a random interval of up to 2 seconds, after the third attempt fails.
- The fifth attempt is made 16 seconds, plus a random interval of up to 4 seconds, after the fourth attempt fails.
- The sixth attempt, and all subsequent attempts are made 25 seconds, plus a random interval of up to 6 seconds and 250 milliseconds after the previous attempt fails.

The reconnection attempts are delayed by intervals that are partly fixed and partly random. This is to prevent all of the IBM MQ classes for JMS applications that were connected to a queue manager that is no longer available from reconnecting simultaneously.

If you need to increase the default values, to more accurately reflect the amount of time that is required for a queue manager to recover, or a standby queue manager to become active, modify the ReconDelay attribute in the Channel stanza of the client configuration file, for more information, see CHANNELS stanza of the client configuration file.

Whether an IBM MQ classes for JMS application continues to work correctly after being reconnected automatically depends on its design. Read the related topics to understand how to design applications can use the automatic reconnection functionality.

Reason codes indicating that a queue manager is no longer available
What reason codes indicate that a queue manager is no longer available, or cannot be reached, when attempting automatic IBM MQ classes for JMS reconnection.

“Automatic JMS client reconnection” on page 275 gives an overview of JMSExceptions and how your applications can restart automatically, and the information in “Using automatic JMS client reconnection” on page 275 details the requirements for automatic client reconnection.

The following information lists the IBM MQ reason codes that your application should check for:

RC2009
MQRC_CONNECTION_BROKEN

RC2059
MQRC_Q_MGR_NOT_AVAILABLE

RC2161
MQRC_Q_MGR_QUIESCING

RC2162
MQRC_Q_MGR_STOPPING

RC2202
MQRC_CONNECTION_QUIESCING

RC2203
MQRC_CONNECTION_STOPPING

RC2223
MQRC_Q_MGR_NOT_ACTIVE

RC2279
MQRC_CHANNEL_STOPPED_BY_USER

RC2537
MQRC_CHANNEL_NOT_AVAILABLE

RC2538
MQRC_HOST_NOT_AVAILABLE

Most JMSExceptions that are thrown back to enterprise applications contain a linked MQException which holds the reason code. To implement the retry logic for the reason codes in the previous list, your enterprise applications should check this linked exception using code similar to the following example:

```
} catch (JMSException ex) {
        Exception linkedEx = ex.getLinkedException();
        if (ex.getLinkedException() != null) {
            if (linkedEx instanceof MQException) {
                MQException mqException = (MQException) linkedEx;
                int reasonCode = mqException.reasonCode;
                // Handle the reason code accordingly
            }
        }
    }
```

Related concepts
IBM MQ classes for JMS

Using automatic client reconnection in Java SE and Java EE environments
You can make use of IBM MQ automatic client reconnection to facilitate various high availability (HA) and disaster recovery (DR) solutions within a Java SE and Java EE environment.

Various HA and DR solutions are available on different platforms:

- Multi Multi-instance queue managers are instances of the same queue manager configured on different servers (see Multi-instance queue managers). One instance of the queue manager is defined as the active instance and another instance is defined as the standby instance. If the active instance fails, the multi-instance queue manager restarts automatically on the standby server.

  Both the active and standby queue managers have the same queue manager identifier (QMID). IBM MQ client applications that connect to a multi-instance queue manager can be configured to automatically reconnect to a standby instance of a queue manager by using automatic client reconnection.

- Linux RDQM (replicated data queue manager) is a high availability solution that is available on Linux platforms (see RDQM high availability). An RDQM configuration consists of three servers configured in a high availability (HA) group, each with an instance of the queue manager. One instance is the running queue manager, which synchronously replicates its data to the other two instances. If the server running this queue manager fails, another instance of the queue manager starts and has current data to operate with. The three instances of the queue manager share a floating IP address, so clients only need to be configured with a single IP address. Client applications that connect to an RDQM queue manager can be configured to automatically reconnect to a standby instance of a queue manager by using automatic client reconnection.

- MQ Appliance An HA solution can also be provided by a pair of IBM MQ Appliances (see High Availability and Disaster Recovery in the IBM MQ Appliance documentation). An HA queue manager runs on one of the appliances, while synchronously replicating data to the standby instance of the queue manager on the other appliance. If the primary appliance fails, the queue manager automatically starts and runs on the other appliance. The two instances of the queue manager can be configured to share a floating IP address, so clients only need to be configured with a single IP address. Client applications that connect to an HA queue manager on an IBM MQ Appliance can be configured to automatically reconnect to the standby instance of a queue manager by using automatic client reconnection.

Note: Within Java EE environments, such as WebSphere Application Server, automatic client reconnection with activation specifications using the functionality provided by IBM MQ classes for JMS is not supported. The IBM MQ resource adapter provides its own mechanism for reconnecting activation specifications if the queue manager that the activation specification was connecting to becomes unavailable. For more information, see "Support for automatic client reconnection in Java EE environments" on page 281.

Related concepts

Multi-instance queue managers

Automatic client reconnection

Related reference

rdqm high availability

Using automatic client reconnection in Java SE environments

Applications using the IBM MQ classes for JMS running in Java SE environments can use the automatic client reconnection functionality through the connection factory property **CLIENTRECONNECTOPTIONS**.

The connection factory property **CLIENTRECONNECTOPTIONS** uses two additional connection factory properties, **CONNECTIONNAMELIST** and **CCDTURL**, to determine how to connect to the server on which the queue manager is running.

CONNECTIONNAMELIST property

The **CONNECTIONNAMELIST** property is a comma-separated list that contains the host name and port information to be used to connect to a queue manager in client mode. This property is used with the **QMANAGER** and **CHANNEL** values. When an application uses the **CONNECTIONNAMELIST** property to create a client connection, the IBM MQ classes for JMS try to connect to each host in list order. If the first queue manager host is unavailable, the IBM MQ classes for JMS attempt to connect to the next host on the list. If the end of the connection name list is reached without creating a connection, the IBM MQ classes for JMS throw the `MQRC_QMGR_NOT_AVAILABLE` IBM MQ reason code.

If the queue manager that the application is connected to fails, any applications that used a **CONNECTIONNAMELIST** to connect to that queue manager receive an exception indicating the queue manager is not available. The application must catch the exception and clear any resources that it was using. To create a connection, the application must use the connection factory. The connection factory attempts to connect to each host in list order again, the queue manager that failed is now not available. The connection factory attempts to connect to another host in the list.

CCDTURL property

The **CCDTURL** property contains a Uniform Resource Locator (URL) that points to a Client Channel Definition Table (CCDT), this property is used with the **QMANAGER** property. The CCDT contains a list of client channels that are used to connect to a queue manager defined on an IBM MQ system. For information on how CCDTs are used by the IBM MQ classes for JMS, see "Using a client channel definition table with IBM MQ classes for JMS" on page 273.

Using the CLIENTRECONNECTOPTIONS property to enable automatic client reconnection within the IBM MQ classes for JMS

The **CLIENTRECONNECTOPTIONS** property is used to enable automatic client reconnection within the IBM MQ classes for JMS. The possible values for this property are as follows:

ASDEF
: The automatic client reconnection behavior is defined by the default value that is specified in the channel stanza of the IBM MQ client configuration file (`mqclient.ini`).

DISABLED
: Automatic client reconnection is disabled.

QMGR
The IBM MQ classes for JMS attempt to connect to a queue manager with the same queue manager identifier as the queue manager that it was connected to, using either of the following options:

- The **CONNECTIONNAMELIST** property and the channel that is defined in the **CHANNEL** property.
- The CCDT defined in the **CCDTURL** property.

ANY
The IBM MQ classes for JMS attempt to reconnect to a queue manager of the same name with the use of either the **CONNECTIONNAMELIST** property or the **CCDTURL**.

Related information
CHANNELS stanza of the client configuration file

Using automatic client reconnection in Java EE environments
The IBM MQ resource adapter, which can be deployed into Java EE (Java Platform, Enterprise Edition) environments, and the WebSphere Application Server IBM MQ messaging provider use the IBM MQ classes for JMS to communicate with IBM MQ queue managers. The IBM MQ resource adapter and the WebSphere Application Server IBM MQ messaging provider provide a number of mechanisms to allow activation specifications, WebSphere Application Server listener ports and applications running inside client containers to automatically reconnect to a queue manager. Enterprise JavaBeans (EJBs) and web-based applications need to implement their own reconnection logic.

Note: Automatic client reconnection with activation specifications using the functionality provided by IBM MQ classes for JMS is not supported (see "Automatic JMS client reconnection" on page 275). The IBM MQ resource adapter provides its own mechanism for reconnecting activation specifications if the queue manager that the activation specification was connecting to becomes unavailable.

The mechanism that the resource adapter provides is controlled by:

- The IBM MQ resource adapter property **reconnectionRetryCount**.
- The IBM MQ resource adapter property **reconnectionRetryInterval**.
- The activation specification property **connectionNameList**.

For more information on these properties, see "Configuration for ResourceAdapter object properties" on page 433.

The use of automatic client reconnection within a message-driven bean application's `onMessage()` method, or any other application that is running within the Java Platform, Enterprise Edition environment is not supported. The application needs to implement its own reconnection logic if the queue manager it was connecting to becomes unavailable. For more information, see "Implementing reconnection logic in a Java EE application" on page 288.

Support for automatic client reconnection in Java EE environments
Within Java EE environments, such as WebSphere Application Server, the IBM MQ resource adapter, and the WebSphere Application Server IBM MQ messaging provider provide a number of mechanisms that allow applications to reconnect to a queue manager automatically. However, in some cases, restrictions apply to this support.

The IBM MQ resource adapter that can be deployed into Java EE environments and the WebSphere Application Server IBM MQ messaging provider, use the IBM MQ classes for JMS to communicate with the IBM MQ queue managers.

The following table summarizes the support that the IBM MQ resource adapter and the WebSphere Application Server IBM MQ messaging provider provide support for automatic client reconnection.

Table 45. Summary of support for automatic client reconnection options in Java EE environments				
Options for automatic reconnection	**CONNECTIONNAMELIST property**	**CCDTURL property**	**CLIENTRECONNECTOPTIONS property**	**Alternative approach to automatic client reconnection**
Activation specifications	Supported with restrictions	Supported with restrictions	Not supported	Java EE environment and activation specifications provide their own reconnection mechanism
WebSphere Application Server listener ports	Supported with restrictions	Supported with restrictions	Not supported	WebSphere Application Server provides its own reconnection mechanism
Enterprise JavaBeans and web-based applications	Supported with restrictions	Supported with restrictions	Not supported	Application must implement its own reconnection logic
Applications running inside client containers	Supported	Supported	Supported	Not applicable

Message-driven bean applications that are installed in a Java EE environment, such as IBM MQ classes for JMS, can use activation specifications to process messages on an IBM MQ system. Activation specifications are used to detect messages that arrive on an IBM MQ system and deliver them to message-driven beans for processing. Message-driven beans can also make more connections to IBM MQ systems from inside their **onMessage()** method. For more information about how these connections can use automatic client reconnection, see Enterprise JavaBeans and Web-based applications.

Activation specifications
For activation specifications, the **CONNECTIONNAMELIST** and **CCDTURL** properties are supported with restrictions and the **CLIENTRECONNECTOPTIONS** property is not supported.

Message-driven bean (MDB) applications that are installed in a Java EE environment, such as WebSphere Application Server, can use activation specifications to process messages on an IBM MQ system.

Activation specifications are used to detect messages arriving on an IBM MQ system, and then deliver them to MDBs for processing. This section deals with how activation specification monitors the IBM MQ system.

MDBs can also make additional connections to IBM MQ systems from inside their `onMessage()` method.

Details about how these connections can use automatic client reconnection can be found in “Enterprise JavaBeans and web-based applications” on page 286.

CONNECTIONNAMELIST property

When starting, the activation specification attempts to connect to the queue manager using the:

- One specified in the **QMANAGER** property
- Channel mentioned in the **CHANNEL** property
- Host name and port information from the first entry in the **CONNECTIONNAMELIST**

If the activation specification is unable to connect to the queue manager using the first entry in the list, the activation specification moves on to the second entry, and so on, until either a connection to the queue manager has been made, or the end of the list has been reached.

If the activation specification is unable to connect to the specified queue manager, using any of the entries in the **CONNECTIONNAMELIST**, the activation specification stops and must be restarted.

Once the activation specification is running, the activation specification gets messages from the IBM MQ system, and delivers the messages to an MDB for processing.

If the queue manager fails while a message is being processed, the Java EE environment detects the failure and attempts to reconnect the activation specification.

The activation specification uses the information in the **CONNECTIONNAMELIST** property as before, when the activation specification performs the reconnection attempts.

If the activation specification tries all of the entries in the **CONNECTIONNAMELIST** and is still unable to connect to the queue manager, then the activation specification waits for the period of time specified by the IBM MQ resource adapter property **reconnectionRetryInterval** before trying again.

The IBM MQ resource adapter property **reconnectionRetryCount** defines the number of consecutive reconnection attempts that are be made before an activation specification is stopped,, and requires a manual restart

Once the activation specification has reconnected to an IBM MQ system, the Java EE environment performs any transactional cleanup that is required, and resumes delivering messages to MDBs for processing.

In order for the transactional cleanup to work correctly, the Java EE environment must be able to access the logs for the queue manager that failed.

If the activation specifications are being used with transactional MDBs that participate in XA transactions, and are connecting to a multi-instance queue manager, the **CONNECTIONNAMELIST** must contain an entry for both the active and standby queue manager instance.

This means that the Java EE environment can access the queue manager logs if the environment needs to perform transaction recovery, regardless of which queue manager the environment reconnects to following a failure.

If the transactional MDBs are being used with stand-alone queue managers, the **CONNECTIONNAMELIST** property must contain a single entry, to ensure that the activation specification always reconnects to the same queue manager running on the same system following a failure.

CCDTURL property

When starting, the activation specification tries to connect to the queue manager specified in the **QMANAGER** property using the first entry in the client channel definition table (CCDT).

If the activation specification is unable to connect to the queue manager using the first entry in the table, the activation specification moves on to the second entry, and so on, until either a connection to the queue manager has been made, or the end of the table has been reached.

If the activation specification is unable to connect to the specified queue manager, using any of the entries in the CCDT, the activation specification stops and must be restarted.

Once the activation specification is running, the activation specification gets messages from the IBM MQ system, and delivers the messages to an MDB for processing.

If the queue manager fails while a message is being processed, the Java EE environment detects the failure and attempts to reconnect the activation specification.

The activation specification uses the information in the CCDT property as before, when the activation specification performs the reconnection attempts.

If the activation specification tries all of the entries in the CCDT and is still unable to connect to the queue manager, the activation specification waits for the period of time specified by the IBM MQ resource adapter property **reconnectionRetryInterval** before trying again.

The IBM MQ resource adapter property **reconnectionRetryCount** defines the number of consecutive reconnection attempts that are be made before an activation specification is stopped,, and requires a manual restart

Once the activation specification has reconnected to an IBM MQ system, the Java EE environment performs any transactional cleanup that is required, and resumes delivering messages to MDBs for processing.

In order for the transactional cleanup to work correctly, the Java EE environment must be able to access the logs for the queue manager that failed.

If the activation specifications are being used with transactional MDBs that participate in XA transactions, and are connecting to a multi-instance queue manager, the CCDT must contain an entry for both the active and standby queue manager instance.

This means that the Java EE environment can access the queue manager logs if the environment needs to perform transaction recovery, regardless of which queue manager the environment reconnects to following a failure.

If the transactional MDBs are being used with stand-alone queue managers, the CCDT must contain a single entry, to ensure that the activation specification always reconnects to the same queue manager running on the same system following a failure.

Ensure that you have set the default value of *PREFERRED* for the **AFFINITY** property on the CCDTs, used with activation specifications, so that connections are made to the same active queue manager.

CLIENTRECONNECTOPTIONS property

Activation specifications provide their own reconnection functionality. The provided functionality allows the specifications to automatically reconnect to an IBM MQ system if the queue manager they were connected to fails.

Because of this, the automatic client reconnection functionality provided by the IBM MQ classes for JMS is not supported.

You must set the **CLIENTRECONNECTOPTIONS** property to *DISABLED* for all activation specifications that are used in the Java EE.

WebSphere Application Server listener ports
Message-driven bean (MDB) applications that are installed in WebSphere Application Server can also use listener ports to process messages on an IBM MQ system.

Listener ports are used to detect messages arriving on an IBM MQ system, and then deliver them to MDBs for processing. This topic explains how the listener port monitors the IBM MQ system.

MDBs can also make additional connections to IBM MQ systems from inside their `onMessage()` method.

See “Enterprise JavaBeans and web-based applications” on page 286 for more information on how these connections can use automatic client reconnection

For WebSphere Application Server listener ports:

- **CONNECTIONNAMELIST** and **CCDTURL** are supported with restrictions
- **CLIENTRECONNECTOPTIONS** is not supported

CONNECTIONNAMELIST property

Listener ports make use of JMS connection pools when connecting to IBM MQ, so are subject to the implications of using connection pools. See “Activation specifications” on page 282 for further information.

If there are no free connections, and the maximum number of connections have not yet been created from this connection factory, the **CONNECTIONNAMELIST** property is used to try and create a new connection to IBM MQ.

If all the IBM MQ systems in the **CONNECTIONNAMELIST** are not accessible, the listener port stops.

The listener port then waits for the period of time specified by the message listener service custom property **RECOVERY.RETRY.INTERVAL** and tries to reconnect again.

This reconnection attempt checks if there are any free connections in the connection pool, just in case one was returned in between the connection attempts. If one is not available, the listener port uses the **CONNECTIONNAMELIST** as before.

Once the listener port has reconnected to an IBM MQ system, the Java EE environment performs any transactional cleanup that is required, and then resumes delivering messages to MDBs for processing.

In order for the transactional cleanup to work correctly, the Java EE environment must be able to access the logs for the queue manager that failed.

If the listener ports are being used with transactional MDBs that participate in XA transactions, and are connecting to a **multi-instance queue manager**, the **CONNECTIONNAMELIST** must contain an entry for both the active and standby queue manager instance.

This means that the Java EE environment can access the queue manager logs if the environment needs to perform transaction recovery, regardless of which queue manager the environment reconnects to following a failure.

If the transactional MDBs are being used with stand-alone queue managers, the **CONNECTIONNAMELIST** property must contain a single entry, to ensure that the activation specification always reconnects to the same queue manager running on the same system following a failure.

CCDTURL property

When starting, the listener port attempts to connect to the queue manager specified in the **QMANAGER** property using the first entry in the CCDT.

If the listener port is not able to connect to the queue manager using the first entry in the table, the listener port moves on to the second entry, and so on, until either a connection to the queue manager has been made, or the end of the table has been reached.

If the listener port is unable to connect to the specified queue manager using any of the entries in the CCDT, the listener port stops.

The listener port then waits for the period of time specified by the message listener service custom property **RECOVERY.RETRY.INTERVAL** and tries to reconnect again.

This reconnection attempt works its way through all of the entries in the CCDT as before.

Once the Listener Port is running, it gets messages from the IBM MQ system and delivers them to an MDB for processing.

If the queue manager fails while a message is being processed, the Java EE environment detects the failure and tries to reconnect the listener port. The listener port uses the information in the CCDT when it performs the reconnection attempts.

If the listener port tries all of the entries in the CCDT and is still unable to connect to the queue manager, then the port waits for the period of time specified by the **RECOVERY.RETRY.INTERVAL** property before trying again.

The message listener service property **MAX.RECOVERY.RETRIES** defines the number of consecutive reconnection attempts that are made before a listener port stops and requires a manual restart.

Once the listener port has reconnected to an IBM MQ system, the Java EE environment performs any transactional cleanup that is required, and then resumes delivering messages to MDBs for processing.

In order for the transactional cleanup to work correctly, the Java EE environment must be able to access the logs for the queue manager that failed.

If the listener ports are being used with transactional MDBs that participate in XA transactions, and are connecting to a multi-instance queue manager, the CCDT must contain an entry for both the active and standby queue manager instance.

This means that the Java EE environment can access the queue manager logs if the environment needs to perform transaction recovery, regardless of which queue manager the environment reconnects to following a failure.

If the transactional MDBs are being used with stand-alone queue managers, the CCDT must contain a single entry, to ensure that the listener port always reconnects to the same queue manager running on the same system following a failure.

Ensure that you have set the default value of *PREFERRED* for the **AFFINITY** property on the CCDTs, used with listener ports, so that connections are made to the same active queue manager.

CLIENTRECONNECTOPTIONS property

Listener ports provide their own reconnection functionality. The provided functionality allows the listener ports to automatically reconnect to an IBM MQ system if the queue manager they were connected to fails.

Because of this, the automatic client reconnection functionality provided by the IBM MQ classes for JMS is not supported.

You must set the **CLIENTRECONNECTOPTIONS** property to *DISABLED* for all listener ports that are used in the Java EE.

Enterprise JavaBeans and web-based applications
Enterprise JavaBean (EJB) applications and applications that run within the web container, such as Servlets, use a JMS connection factory to create a connection to an IBM MQ queue manager.

The following restrictions apply to EJBs and web-based applications:

- **CONNECTIONNAMELIST** and **CCDTURL** are supported with restrictions
- **CLIENTRECONNECTOPTIONS** is not supported

CONNECTIONNAMELIST property

If the Java EE environment provides a connection pool for JMS connections, see “Using CONNECTIONNAMELIST or CCDT in a connection pool” on page 288 for information on how this affects the behavior of the **CONNECTIONNAMELIST** property.

If the Java EE environment does not provide a JMS connection pool. the application uses the **CONNECTIONNAMELIST** property in the same way as Java SE applications.

If the applications are being used with transactional MDBs that participate in XA transactions, and are connecting to a multi-instance queue manager, the **CONNECTIONNAMELIST** must contain an entry for both the active and standby queue manager instance.

This means that the Java EE environment can access the queue manager logs if the environment needs to perform transaction recovery, regardless of which queue manager the environment reconnects to following a failure.

If the applications are being used with stand-alone queue managers, the **CONNECTIONNAMELIST** property must contain a single entry, to ensure that the application always reconnects to the same queue manager, running on the same system, following a failure.

CCDTURL property

If the Java EE environment provides a connection pool for JMS connections, see “Using CONNECTIONNAMELIST or CCDT in a connection pool” on page 288 for information on how this affects the behavior of the **CCDTURL** property.

If the Java EE environment does not provide a JMS connection pool. the application uses the **CCDTURL** property in the same way as Java SE applications.

If the applications are being used with transactional MDBs that participate in XA transactions, and are connecting to a multi-instance queue manager, the CCDT must contain an entry for both the active and standby queue manager instance.

This means that the Java EE environment can access the queue manager logs if the environment needs to perform transaction recovery, regardless of which queue manager the environment reconnects to following a failure.

If the applications are being used with stand-alone queue managers, the CCDT must contain a single entry, to ensure that the activation specification always reconnects to the same queue manager running on the same system following a failure.

CLIENTRECONNECTOPTIONS property

You must set the **CLIENTRECONNECTOPTIONS** property to *DISABLED* for all JMS connection factories used by EJBs, or applications, that run in the web container.

Applications that require to automatically reconnect to a new queue manager, if the queue manager they are using fails, need to implement their own reconnection logic. See "Implementing reconnection logic in a Java EE application" on page 288 for more information.

Scenarios: WebSphere Application Server with IBM MQ
Scenarios: WebSphere Application Server Liberty profile with IBM MQ

Applications running inside client containers
Some Java EE environments, such as WebSphere Application Server, provide a client container that can be used to run Java SE applications.

Applications running inside of these environments use a JMS connection factory to connect to an IBM MQ queue manager.

For applications running inside client containers:

- **CONNECTIONNAMELIST** and **CCDTURL** are fully supported
- **CLIENTRECONNECTOPTIONS** is fully supported

CONNECTIONNAMELIST property

If the Java EE environment provides a connection pool for JMS connections, see "Using CONNECTIONNAMELIST or CCDT in a connection pool" on page 288 for information on how this affects the behavior of the **CONNECTIONNAMELIST** property.

If the Java EE environment does not provide a JMS connection pool. the application uses the **CONNECTIONNAMELIST** property in the same way as Java SE applications.

CCDTURL property

If the Java EE environment provides a connection pool for JMS connections, see "Using CONNECTIONNAMELIST or CCDT in a connection pool" on page 288 for information on how this affects the behavior of the **CCDTURL** property.

If the Java EE environment does not provide a JMS connection pool. the application uses the **CCDTURL** property in the same way as Java SE applications.

Using CONNECTIONNAMELIST or CCDT in a connection pool
Some Java EE environments, for example, WebSphere Application Server, provide a JMS connection pool. container that can be used to run Java SE applications.

Applications that create a connection using a connection factory that has been defined in the Java EE environment either obtain an existing free connection from the connection pool for this connection factory, or a new connection if there is not a suitable one in the connection pool.

This can have implications if the connection factory has been configured with either the **CONNECTIONNAMELIST** or **CCDTURL** property defined.

The first time the connection factory is used to create a connection, the Java EE environment uses either the **CONNECTIONNAMELIST**. or the **CCDTURL** to create a new connection to the IBM MQ system. When this connection is no longer required, it is returned to the connection pool where the connection becomes available for reuse.

If something else creates a connection from the connection factory, the Java EE environment returns the connection from the connection pool, rather than using the **CONNECTIONNAMELIST** or the **CCDTURL** properties to create a new connection.

If a connection is being used when a queue manager instance fails, the connection is discarded. However, the contents of the connection pool might not be, which means that the pool can potentially still contain connections to a queue manager that is no longer running.

In this situation, the next time a request is made to create a connection from the connection factory, a connection to the failed queue manager is returned. Any attempts to use this connection fail, as the queue manager is no longer running, causing the connection to be discarded.

Only when the connection pool is empty, will the Java EE environment use the **CONNECTIONNAMELIST** or **CCDTURL** properties to create a new connection to IBM MQ.

Due to the way the **CONNECTIONNAMELIST** and CCDTs are used to create JMS connections, it is also possible to have a connection pool that contains connections to different IBM MQ systems.

For example, assume that a connection factory has been configured with the **CONNECTIONNAMELIST** property set to the following value:

```
CONNECTIONNAMELIST = hostname1(port1), hostname2(port2)
```

Suppose that the first time an application tries to create a connection to a stand-alone queue manager from this connection factory, the queue manager running on the system `hostname1(port1)` is not accessible. This means that the application ends up with a connection to the queue manager running on `hostname2(port2)`.

Another application now comes along and creates a JMS connection from the same connection factory. The queue manager on `hostname1(port1)` is now available, so a new JMS connection is created to this IBM MQ system and is returned to the application.

When both applications have finished, they close their JMS Connections, which causes the connections to be returned to the connection pool.

The result is that the connection pool for our connection factory now contains two JMS connections:

- One connection to the queue manager running on `hostname1(port1)`
- One connection to the queue manager running on `hostname2(port2)`

This can lead to issues relating to transaction recovery. If the Java EE system needs to roll back a transaction, it needs to be able to connect to a queue manager that has access to the transaction logs.

Implementing reconnection logic in a Java EE application
Enterprise JavaBeans and web-based applications that want to automatically reconnect if a queue manager fails need to implement their own reconnection logic.

The following options give more information on how you might achieve this.

Allow the application to fail

This approach requires no application changes, but does require an administrative reconfiguration of the connection factory definition to include the **CONNECTIONNAMELIST** property. However, this approach does require the invoker to be able to handle a failure appropriately . Note that this is also required for failures such as MQRC_Q_FULL that are not related to connection failure.

Example code for this process:

```
public class SimpleServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
         try {
 // get connection factory/ queue
 InitialContext ic = new InitialContext();
 ConnectionFactory cf =
           (ConnectionFactory)ic.lookup("java:comp/env/jms/WMQCF");
 Queue q = (Queue) ic.lookup("java:comp/env/jms/WMQQueue");

 // send a message
 Connection c = cf.createConnection();
 Session s = c.createSession(false, Session.AUTO_ACKNOWLEDGE);
 MessageProducer p = s.createProducer(q);
 Message m = s.createTextMessage();
 p.send(m);

 // done, release the connection
 c.close();
 }
 catch (JMSException je) {
 // process exception
   }
 }
}
```

The preceding code assumes that the connection factory, this servlet is using, has the **CONNECTIONNAMELIST** property defined.

When the servlet first processes, a new connection is created using the **CONNECTIONNAMELIST** property, assuming that no pooled connections are available from other applications connecting to the same queue manager.

When the connection is released following a `close()` call, this connection is returned to the pool and reused the next time the servlet runs - without referring to the **CONNECTIONNAMELIST** - until a connection failure occurs, at which point a CONNECTION_ERROR_OCCURRED event is generated. This event prompts the pool to destroy the failed connection.

When the application next runs, no pooled connection is available and the **CONNECTIONNAMELIST** is used to connect to the first available queue manager. If queue manager fail over has taken place (for example, the failure was not a transitory network failure) the servlet connects to the backup instance once it is available.

If other resources, such as databases, are involved in the application, it might be appropriate to indicate that the application server should roll back the transaction.

Handle reconnection within the application

If the invoker is unable to process a failure from the servlet, then reconnection must be handled within the application. As shown in the following example, to handle a reconnection within the application requires the application to request a new connection so that it can cache the connection factory that it looked up from JNDI and handle a `JMSException` such as JMSCMQ0001: `WebSphere MQ call failed with compcode '2' ('MQCC_FAILED') reason '2009' ('MQRC_CONNECTION_BROKEN')`.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
```

```
  // get connection factory/ queue
  InitialContext ic = new InitialContext();
  ConnectionFactory cf = (ConnectionFactory)
              ic.lookup("java:comp/env/jms/WMQCF");
  Destination destination = (Destination) ic.lookup("java:comp/env/jms/WMQQueue");

  setupResources();

  // loop sending messages
  while (!sendComplete) {
    try {
      // create the next message to send
      msg.setText("message sent at "+new Date());
      // and send it
      producer.send(msg);
    }
    catch (JMSException je) {
        // drive reconnection
        setupResources();
    }
  }
```

In the following example, `setupResources()` creates the JMS objects and includes a sleep and retry loop to handle non-instantaneous reconnection. In practice, this method prevents many reconnect attempts. Note that exit conditions have been omitted from the example for clarity.

```
 private void setupResources() {

    boolean connected = false;
    while (!connected) {
      try {
        connection = cf.createConnection(); // cf cached from JNDI lookup
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        msg = session.createTextMessage();
        producer = session.createProducer(destination); // destination cached from JNDI lookup
        // no exception? then we connected ok
        connected = true;
      }
      catch (JMSException je) {
        // sleep and then have another attempt
        try {Thread.sleep(30*1000);} catch (InterruptedException ie) {}
      }
    }
```

If the application manages reconnection, it is important that the application releases any connections that are held to other resources, whether these resources are other IBM MQ queue managers or other back end services such as databases. You must reestablish these connections when reconnection to a new IBM MQ queue manager instance is complete. If you do not do reestablish the connections, application server resources are held unnecessarily during the reconnection attempt, and might have timed out by the time they are reused.

Use of the WorkManager

For long-lived applications (for example, batch processing) where processing time is greater than a few tens of seconds, the WebSphere Application Server WorkManager can be used. A code fragment example for WebSphere Application Server follows:

```
public class BatchSenderServlet extends HttpServlet  {

  private WorkManager workManager = null;
  private MessageSender sender; // background sender WorkImpl

  public void init() throws ServletException {
    InitialContext ctx = new InitialContext();
    workManager = (WorkManager)ctx.lookup(java:comp/env/wm/default);
    sender = new MessageSender(5000);
    workManager.startWork(sender);
  }

  public void destroy() {
    sender.halt();
  }
```

```
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                 throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    if (sender.isRunning()) {
      out.println(sender.getStatus());
    }
}
```

where `web.xml` contains:

```
<resource-ref>
      <description>WorkManager</description>
      <res-ref-name>wm/default</res-ref-name>
      <res-type>com.ibm.websphere.asynchbeans.WorkManager</res-type>
      <res-auth>Container</res-auth>
      <res-sharing-scope>Shareable</res-sharing-scope>
   </resource-ref>
```

and the batch is now implemented through the work interface:

```
import com.ibm.websphere.asynchbeans.Work;

public class MessageSender implements Work {

  public MessageSender(int messages) {numberOfMessages = messages;}

  public void run() {
    // get connection factory/ queue
    InitialContext ic = new InitialContext();
    ConnectionFactory cf = (ConnectionFactory)
                ic.lookup("java:comp/env/jms/WMQCF");
    Destination destination = (Destination) ic.lookup("jms/WMQQueue");

    setupResources();

    // loop sending messages
    while (!sendComplete) {
      try {
        // create the next message to send
        msg.setText("message sent at "+new Date());
        // and send it
        producer.send(msg);
        // are we finished?
        if (sendCount == numberOfMessages) {sendComplete = true);
      }
      catch (JMSException je) {
          // drive reconnection
          setupResources();
      }
  }

  public boolean isRunning() {return !sendComplete;}

  public void release() {sendComplete = true;}
```

If the batch processing takes a long time to run, for example, large messages, slow network, or extensive database access (especially when coupled with slow fail over) then the server starts to output hung thread warnings, similar to the following example:

WSVR0605W: Thread "`WorkManager.DefaultWorkManager` : 0" (00000035) has been active for 694061 milliseconds and may be hung. There is/are 1 thread(s) in total in the server that may be hung.

These warnings can be minimized by reducing the batch size, or increasing the hung thread timeout. However, it is generally preferable if you implement this processing in an EJB (for batch send) or message-driven bean (for consume or consume and reply) processing.

Note that application-managed reconnection does not provide a general solution to handling run time errors, and the application must still handle errors that are not related to connection failure.

For example, attempting to put a message to a queue that is full (2053 MQRC_Q_FULL), or attempting to connect to a queue manager using security credentials that are not valid (2035 MQRC_NOT_AUTHORIZED).

The application must also handle 2059 MQRC_Q_MGR_NOT_AVAILABLE errors when no instances are immediately available when fail over is in progress. This can be achieved by the application reporting the JMS exceptions as they occur, instead of silently attempting to reconnect.

IBM MQ classes for JMS object pooling
Using a form of connection pooling outside of Java EE helps to reduce overall load resulting, for example, from some stand-alone applications using frameworks, or being deployed into cloud environments, and also from a greater number of client connections into `QueueManagers` leading to an increase in server consolidation of applications and queue managers

Within the Java EE programming model, there is a well defined life cycle of the various objects in use. Message-driven beans (MDBs) are most constrained, whilst Servlets provide more freedom. Therefore, the pooling options that are available within the Java EE servers suit the various programming models used.

With Java SE (or with another framework such as Spring) the programming models are extremely flexible. Therefore a single pooling strategy does not suit all. You should consider if there is going to a framework in place that could do any form of pooling, for example, Spring.

The pooling strategy to use depends on the environment in which your application is running.

Object pooling in a Java EE environment
Java EE application servers provide connection pooling functionality that can be used by message-driven bean applications, Enterprise Java Beans and Servlets.

WebSphere Application Server maintains a pool of connections to a JMS provider, in order to improve performance. When an application creates a JMS connection, the application server determines if a connection already exists in the free connection pool. If so, the connection is returned to the application; otherwise, a new connection is created.

Figure 41 on page 292 shows how both activation specifications and listener ports establish a JMS connection and use that connection to monitor a destination for messages in normal mode.

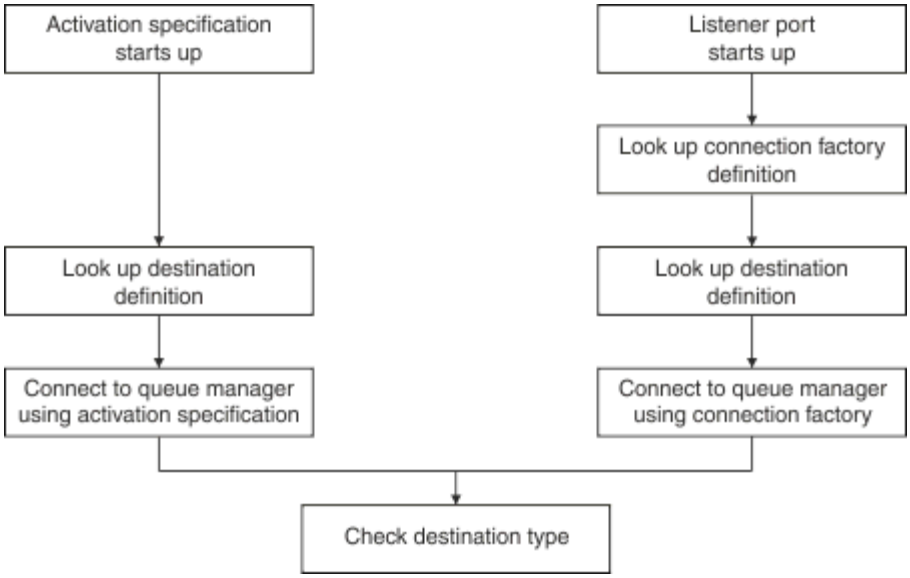


Figure 41. Normal mode

When using the IBM MQ messaging provider, applications that perform outbound messaging (such as enterprise Java Beans and servlets), and the message-driven bean listener port component, can make use of these connection pools.

IBM MQ messaging provider activation specifications use connection pooling functionality provided by the IBM MQ resource adapter. See Configuring properties for the WebSphere MQ resource adapter for more information.

"Examples of using the connection pool" on page 297 explains how applications that perform outbound messaging, and listener ports, use the free pool when creating JMS connections.

"Free connection pool maintenance threads" on page 299 explains what happens to these connections when an application, or listener port, has finished with the connections.

"Pool maintenance thread examples" on page 301 explains how the free connection pool is cleaned to prevent JMS connections from becoming stale.

WebSphere Application Server has a limit on the number of connections that can be created from a factory, specified by the *maximum connections* property of the Connection Factory. The default value for this property is 10, which means there can be up to 10 connections created from a factory at any one time.

Each factory has an associated free connection pool. When the application server starts up, the free connection pools are empty. The maximum number of connections that can exist in the free pool for a factory is also specified by the Maximum connections property.

Tip: With JMS 2.0, a connection factory can be used to create both connections and contexts. As a result, it is possible to have a connection pool associated with a connection factory that contains a mixture of both connections and contexts. It is recommended that a connection factory is only used for creating connections or creating contexts. This ensures that the connection pool for that connection factory only contains objects of a single type, which makes the pool more efficient.

For information about how connection pooling works in WebSphere Application Server, see Configuring connection pooling for JMS connections. For other application servers, refer to the appropriate application server documentation.

How the connection pool is used

Every JMS connection factory has a connection pool associated with it, and the connection pool contains zero or more JMS connections. Every JMS connection has an associated JMS session pool, and every JMS session pool contains zero or more JMS sessions.

Figure 42 on page 294 shows the relationship between these objects.

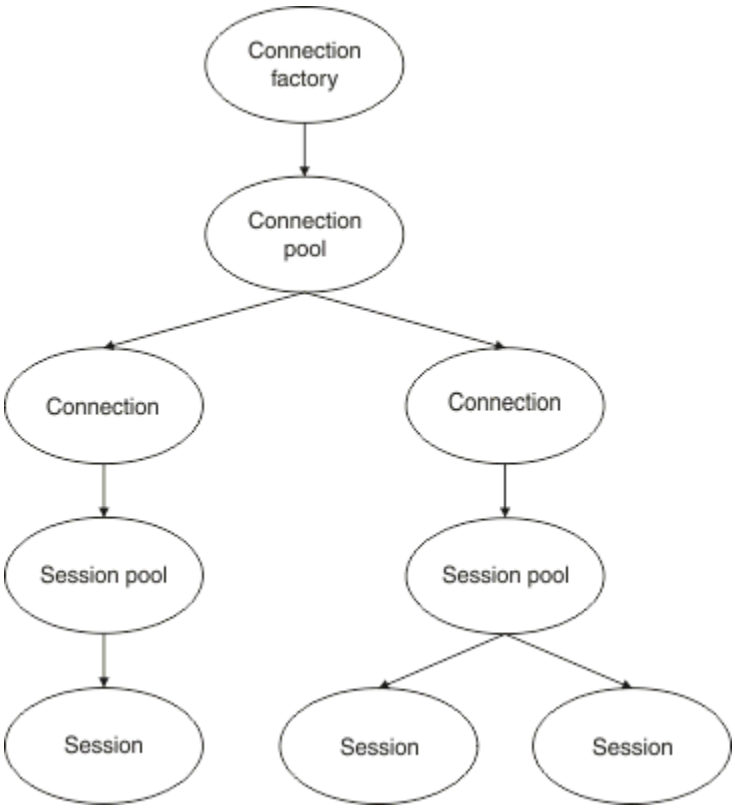


Figure 42. Connection pools and session pools

When a listener port starts up, or an application that wants to do outbound messaging uses the factory to create a connection, the port or application calls one of the following methods:

- **`connectionFactory.createConnection()`**
- **`ConnectionFactory.createConnection(String, String)`**
- **`QueueConnectionFactory.createQueueConnection()`**
- **`QueueConnectionFactory.createQueueConnection(String, String)`**
- **`TopicConnectionFactory.createTopicConnection()`**
- **`TopicConnectionFactory.createTopicConnection(String, String)`**

The WebSphere Application Server connection manager tries to obtain a connection from the free pool for this factory, and return it to the application.

If there are no free connections in the pool, and the number of connections created from this factory has not reached the limit specified in the *maximum connections* property of that factory, the Connection Manager creates a new connection for the application to use.

However, if an application attempts to create a connection, but the number of connections created from this factory already is equal to the *maximum connections* property of the factory, the application waits for a connection to become available (to be put back in the free pool).

The time that the application waits is specified in the *connection timeout* property of the connection pool, which has a default value of 180 seconds. If a connection is put back in the free pool within this 180 second period, the Connection Manager immediately takes it out of the pool again and passes it to the application. However, if the timeout period elapses, a *ConnectionWaitTimeoutException* is thrown.

When an application has finished with the connection and closes it by calling:

- **`Connection.close()`**
- **`QueueConnection.close()`**
- **`TopicConnection.close()`**

the connection is actually kept open, and is returned to the free pool so that it can be reused by another application. Therefore, you can have connections open between WebSphere Application Server and the JMS provider, even if no JMS applications are running on the application server.

Advanced connection pool properties
There are a number of advanced properties that can be used to control the behavior of JMS connection pools.

Surge protection

"How applications that perform outbound messaging use the connection pool" on page 298 describes the use of the `sendMessage()` method, which incorporates `connectionFactory.createConnection()`.

Consider the situation where you have 50 EJBs all creating JMS connections from the same connection factory as part of their `ejbCreate()` method.

If all of these beans are created at the same time, and there are no connections in the free connection pool of the factory, the application server tries to create 50 JMS connections to the same JMS provider simultaneously. The result is a significant load on both WebSphere Application Server and the JMS provider.

The surge protection properties can prevent this situation by limiting the number of JMS connections that can be created from a connection factory at any one time, and staggering the creation of additional connections.

Limiting the number of JMS connections at any one time is achieved by using two properties:

- Surge threshold
- Surge creation interval.

When EJB applications try to create a JMS connection from a connection factory, the connection manager checks to see how many connections are being created. If that number is less than or equal to the value of the `surge threshold` property, the connection manager continues opening new connections.

However, if the number of connections being created exceeds the `surge threshold` property, then the connection manager waits for the period of time specified by the `surge creation interval` property before creating and opening a new connection.

Stuck connections

A JMS connection is considered `stuck`, if a JMS application uses that connection to send a request to the JMS provider, and the provider does not respond within a certain amount of time.

WebSphere Application Server provides a way to detect `stuck` JMS connections To use this function, you must set three properties:

- Stuck Time Timer
- Stuck Time
- Stuck Threshold

"Pool maintenance thread examples" on page 301 explains how the pool maintenance thread runs periodically and checks the contents of the free pool of a connection factory, looking for connections that have either gone unused for a period of time, or have been in existence for too long.

To detect stuck connections, the application server also manages a stuck connection thread that checks the state of all active connections created from a connection factory to see if any of them are waiting for a reply from the JMS provider.

When the stuck connection thread runs is determined by the `Stuck time timer` property. The default value for this property is zero, which means that stuck connection detection never runs.

If the thread finds one waiting for a response, it determines how long it has been waiting, and compares this time to the value of the `Stuck time` property.

If the time taken for the JMS provider to respond exceeds the time specified by the `Stuck time` property, the application server marks the JMS connection as stuck.

For example, suppose the connection factory `jms/CF1` has the `Stuck time timer` property set to 10, and the `Stuck time` property set to 15.

The stuck connection thread becomes active every 10 seconds, and checks if any connection created from `jms/CF1` has been waiting for longer than 15 seconds for a response from IBM MQ.

Suppose an EJB creates a JMS connection to IBM MQ using `jms/CF1`, and then tries to create a JMS Session using that connection by calling `Connection.createSession()`.

However, something is preventing the JMS provider from responding to the request. Perhaps the machine has frozen, or a process running on the JMS provider is deadlocked, preventing any new work from being processed:

Ten seconds after the EJB called `Connection.createSession()`, the stuck connection timer becomes active, and looks at the active connections created from `jms/CF1`.

Assume there is only one active connection, for example called `c1`. The first EJB has been waiting 10 seconds for a response to a request it sent down to `c1`, which is less than the value of `Stuck time`, so the stuck connection timer ignores this connection and becomes inactive.

10 seconds later, the stuck connection thread becomes active again, and looks at the active connections for `jms/CF1`. As before, assume that there is only the one connection, `c1`.

It is now 20 seconds since the first EJB called `createSession()`, and the EJB is still waiting for a response. 20 seconds is longer than the time specified in the `Stuck time` property, so the stuck connection thread marks `c1` as stuck.

If, five seconds later, IBM MQ finally responds, and allows the first EJB to create a JMS Session, the connection is back in use.

The application server counts the number of JMS connections created from a connection factory that are stuck. When an application uses that connection factory to create a new JMS Connection, and there are no free connections in the free pool of that factory, the connection manager compares the number of stuck connections to the value of the `Stuck threshold` property.

If the number of stuck connections is less than the value set for the `Stuck threshold`, property, the connection manager creates a new connection and gives it to the application.

However, if the number of stuck connections is equal to the value of the `Stuck threshold` property, the application gets a resource exception.

Pool partitions

WebSphere Application Server provides two properties that let you partition the free connection pool for a connection factory:

- `Number of free pool partitions` tells the application server how many partitions you want to divide the free connection pool into.
- `Free pool distribution table size` determines how the partitions are indexed.

Leave these properties at their default values of zero, unless you are asked to change them by your IBM Support Center.

Note that WebSphere Application Server has one additional advanced connection pool property called `Number of shared partitions`. This property specifies the number of partitions used to store shared connections. However, as JMS connections are always unshared, this property does not apply.

Examples of using the connection pool
The message-driven bean listener port component, and applications that perform outbound messaging, use a JMS connection pool.

Figure 43 on page 297 shows how the connection pool works for WebSphere Application Server V7.5 and V8.0.

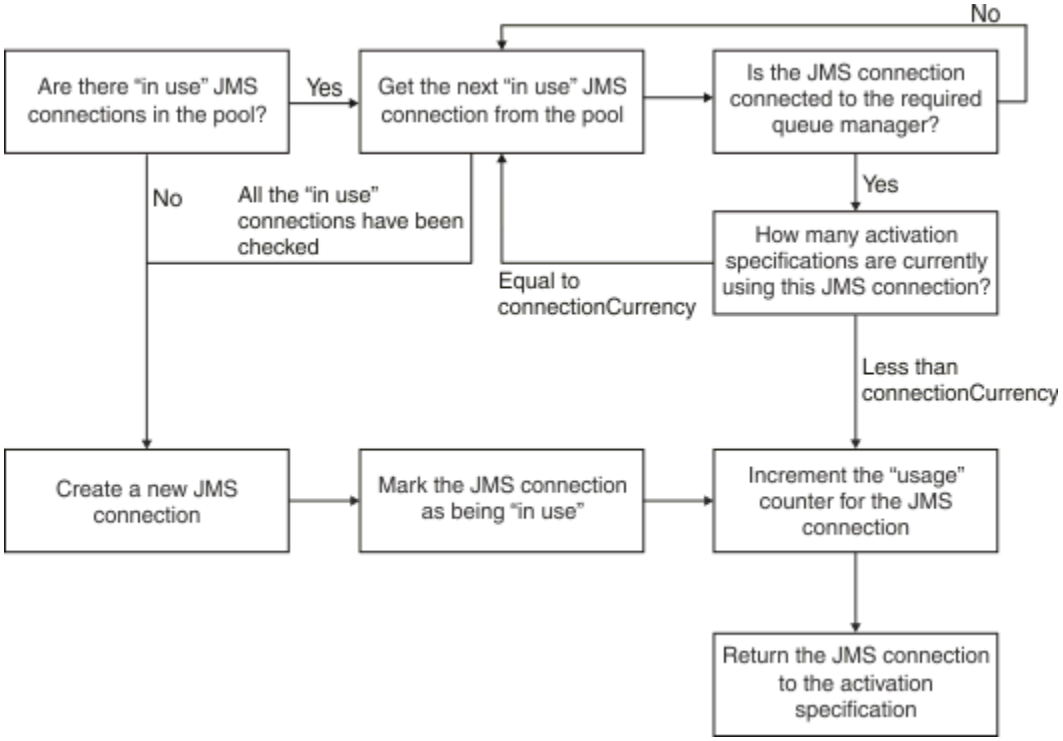


Figure 43. WebSphere Application Server V7.5 and V8.0 - how the connection pool works

Figure 44 on page 297 shows how the connection pool works for WebSphere Application Server V8.5.

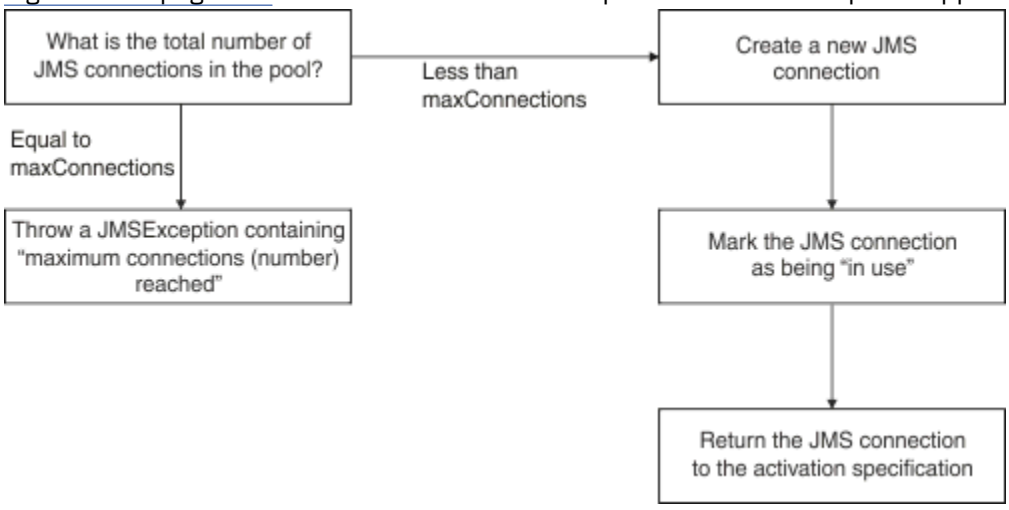


Figure 44. WebSphere Application Server V8.5 - how the connection pool works

How MDB listener ports use the connection pool

Assume you have a MDB deployed on a WebSphere Application Server Network Deployment system, that is using IBM MQ as the JMS provider. The MDB is deployed against a listener port which is using a connection factory called, for example, `jms/CF1`, that has the *maximum connections* property set to 2, which means that only two connections can be created from this factory at any one time.

When the listener port starts up, the port attempts to create a connection to IBM MQ, using the`jms/CF1` connection factory.

To do so, the port requests a connection from the connection manager. Since this is the first time the `jms/CF1` connection factory has been used, there are no connections in the `jms/CF1` free connection pool, so the connection manager creates a new one called, for example, `c1`. Note, that this connection exists for the entire life of the listener port.

Now, consider the situation where you stop the listener port using the WebSphere Application Server administrative console. In this case, the connection manager takes the connection, and puts it back into the free pool. However, the connection to IBM MQ remains open.

If you restart the listener port, the port once again asks the connection manager for a connection to the queue manager. As you now have a connection (`c1`) in the free pool, the connection manager takes this connection out of the pool and makes it available to the listener port.

Now, assume that you have a second MDB deployed into the application server, and it is using a different listener port.

Suppose you then try to start a third listener port, that is also configured to use the `jms/CF1` connection factory. The third listener port requests a connection from the connection manager, which looks in the free pool for `jms/CF1` and finds that it is empty. It then checks how many connections have already been created from the `jms/CF1` factory.

Since the maximum connections property for `jms/CF1` is set to 2, and you have already created two connections from this factory, the connection manager waits for 180 seconds (the default value of the connection timeout property) for a connection to become available.

However, if you stop the first listener port, its connection `c1` is put into the free pool for `jms/CF1`. The connection manager retrieves this connection and gives it to the third listener.

If you now try to restart the first listener, this listener has to wait for one of the other listener ports to stop before the first listener can restart. If none of the running listener ports is stopped within 180 seconds, the first listener receives a `ConnectionWaitTimeoutException` error and stops.

How applications that perform outbound messaging use the connection pool

For this option, suppose that there is a single EJB called, for example `EJB1`, installed in the application server. The bean implements a method called `sendMessage()` by:

- Creating a JMS connection to IBM MQ from a factory `jms/CF1`, using `connectionFactory.createConnection()`.
- Creating a JMS session from the connection.
- Creating a message producer from the session.
- Sending a message.
- Closing the producer.
- Closing the session.
- Closing the connection, by calling `connection.close()`.

Assume that the free pool for the factory `jms/CF1` is empty. When the EJB is invoked for the first time, the bean attempts to create a connection to IBM MQ from the factory `jms/CF1`. As the free pool for the factory is empty, the connection manager creates a new connection and gives it to `EJB1`.

Just before the method exits, the method calls `connection.close()`. Rather than closing `c1`, the connection manager takes the connection and puts it into the free pool for `jms/CF1`.

The next time `sendMessage()` is called, the `connectionFactory.createConnection()` method returns `c1` to the application.

Assume that you have a second instance of the EJB running at the same time as the first instance. When both instances are calling `sendMessage()`, two connections are created from the `jms/CF1` connection factory.

Now assume that a third instance of the bean is created. When the third bean invokes `sendMessage()`, the method calls `connectionFactory.createConnection()` to create a connection from `jms/CF1`.

However, there are currently two connections created from `jms/CF1`, which equals the value of maximum connections for this factory. Therefore, the `createConnection()` method waits for 180 seconds (the default value of the connection timeout property) for a connection to become available.

However, if the `sendMessage()` method for the first EJB calls `connection.close()` and exits, the connection it was using, `c1`, is put back into the free connection pool. The connection manager takes the connection back out of the free pool and gives it to the third EJB. The call from that bean to `connectionFactory.createConnection()` then returns, allowing the `sendMessage()` method to complete.

MDB listener ports and EJBs using the same connection pool

The two previous examples show how listener ports and EJBs can use the connection pool in isolation. However, you can have both a listener port and an EJB running inside the same application server and creating JMS connections using the same connection factory.

You need to consider the implications of this situation

The key thing to remember is that the connection factory is shared between the listener port and the EJB.

For example, assume that you have a listener and EJB running at the same time. Both are using the `jms/CF1` connection factory, which means that the connection limit specified by the maximum connections property for that factory has been reached.

If you try to start either another listener port, or another instance of an EJB, either has to wait for a connection to be returned to the free connection pool for `jms/CF1`.

Free connection pool maintenance threads
Associated with each free connection pool is a pool maintenance thread, which monitors the free pool to ensure that the connections in it are still valid.

If the pool maintenance thread decides that a connection in the free pool needs to be discarded, the thread physically closes the JMS connection to IBM MQ.

How the pool maintenance thread works

The behavior of the pool maintenance thread is determined by the value of four properties of the connection pool:

Aged timeout
The amount of time a connection remains open.

Minimum connections
The minimum number of connections the connection manager keeps in the free pool of a connection factory.

Reap time
How often the pool maintenance thread runs.

Unused timeout
How long a connection remains in the free pool before it is closed.

By default, the pool maintained thread runs every 180 seconds, although this value can be changed by setting the connection pool **`Reap time`** property.

The maintenance thread looks at each connection in the pool, checks how long it has been in the pool, and how much time has elapsed since it was created and last used.

If the connection has not been used for a period longer than the value of the **`Unused timeout`** property for the connection pool, the maintenance thread checks the number of connections currently in the free pool. If that number is:

- Greater than the value of **`Minimum connections`**, the connection manager closes the connection.

- Equals the value of **Minimum connections**, the connection is not closed and remains in the free pool.

The default value of the **Minimum connections** property is *1*, which means that, for performance reasons, the connection manager always attempts to keep at least one connection in the free pool.

The **Unused timeout** property has a default value of 1800 seconds. By default, if a connection is put back in the free pool and not used again for at least 1800 seconds, that connection is closed, provided that closing it, leaves at least one connection in the free pool.

This procedure prevents unused connections from becoming stale. To turn this feature off, set the **Unused timeout** property to zero.

If a connection is in the free pool, and the elapsed time since its creation is greater than the value of the **Aged timeout** property for the connection pool, then it is closed regardless of how long it has been since it was last used.

By default, the **Aged timeout** property is set to zero, which means that the maintenance thread never performs this check. Connections that have been around for longer than the **Aged timeout** property are discarded regardless of how many connections will remain in the free pool. Note that the **Minimum connections** property has no affect in this situation.

Disabling the pool maintenance thread

From the previous description, you can see that the pool maintenance thread does a great deal of work when active, particularly if there are a large number of connections in the free pool of the connection factory.

For example, suppose there are three JMS connection factories, with the **Maximum connections** property set to 10 for each factory. Every 180 seconds, three pool maintenance threads become active and scan the free pools for each connection factory respectively. If the free pools have many connections, the maintenance threads have much work to do, which can significantly impact performance.

You can disable the pool maintenance thread for an individual free connection pool by setting its **Reap time** property to zero.

Disabling the maintenance thread means that connections are never closed, even if the **Unused timeout** has elapsed. However, the connections can still be closed if the **Aged timeout** has passed.

When an application has finished with a connection, the connection manager checks to see how long the connection has existed, and if that period is longer than the value of the **Aged timeout** property, the connection manager closes the connection rather than returning it to the free pool.

Transactional implications of Aged timeout

As described in the previous section, the **Aged timeout** property specifies how long a connection to the JMS provider remains open before the connection manager closes it.

The default value for the **Aged timeout** property is zero, which means that the connection will never be closed because it is too old. You should leave the **Aged timeout** property at this value, because enabling **Aged timeout** can have transactional implications when using JMS inside of EJBs.

In JMS, the unit of a transaction is a JMS *session*, which is created from a JMS *connection*. It is the JMS *session* that is enlisted into transactions, and not the JMS *connection*.

Due to the design of the application server, JMS connections can be closed because the **Aged timeout** has elapsed, even if JMS sessions created from that connection are involved in a transaction.

Closing a JMS connection causes any outstanding transactional work on JMS sessions to be rolled back, as described in the JMS specification. However, the application server is unaware that the JMS sessions created from the connection are no longer valid. When the server tries to use the session to commit or roll back a transaction, an `IllegalStateException` occurs.

Important: If you want to use **Aged timeout** with JMS connections from within EJBs, ensure that any JMS work is explicitly committed on the JMS session, before the EJB method that performs the JMS operations exits.

Pool maintenance thread examples
Using the Enterprise JavaBean (EJB) example to understand how the pool maintenance thread works. Note that you can also use Message Driven Beans (MDBs) and listener ports, as all you need is a way to get connections in the free pool.

See "How applications that perform outbound messaging use the connection pool" on page 298 for further details of the `sendMessage()` method.

You have configured the connection factory with the following values:

- **`Reap time`** at its default value of 180 seconds
- **`Aged timeout`** at its default value of zero seconds
- **`Unused timeout`** set to 300 seconds

After the application server starts up, the `sendMessage()` method is invoked.

The method creates a connection called, for example `c1`, using the factory `jms/CF1`, uses that factory to send a message, and then calls `connection.close()`, which causes `c1` to be put into the free pool.

After 180 seconds, the pool maintenance thread starts up, and looks at the `jms/CF1` free connection pool. There is a free connection `c1` in the pool, so the maintenance thread looks at the time the connection was put back, and compares this to the current time.

180 seconds have passed since the connection was put in the free pool, which is less than the value of the **`Unused timeout`** property for `jms/CF1`. Therefore the maintenance thread leaves the connection alone.

180 seconds later, the pool maintenance thread runs again. The maintenance thread finds the connection `c1`, and determines that the connection has been in the pool for 360 seconds, which is longer than the **`Unused timeout`** value set, so the connection manager closes the connection.

If you now run the `sendMessage()` method again, when the application calls `connectionFactory.createConnection()` , the connection manager creates a new connection to IBM MQ because the free connection pool for the connection factory is empty.

The preceding example shown how the maintenance thread uses the **`Reap time`** and **`Unused timeout`** properties to prevent stale connections, when the **`Aged timeout`** property is set to zero.

How does the **`Aged timeout`** property work?

In the following example, assume that you have set the:

- **`Aged timeout`** property to 300 seconds
- **`Unused timeout`** property to zero.

You invoke the `sendMessage()` method and this method tries to create a connection from the `jms/CF1` connection factory.

As the free pool for this factory is empty, the connection manager creates a new connection, `c1`, and returns it to the application. When `sendMessage()` calls `connection.close()`, `c1` is put back into the free connection pool.

180 seconds later, the pool maintenance thread runs. The thread finds `c1` in the free connection pool, and checks how long ago it was created. The connection has existed for 180 seconds, which is less than **`Aged timeout`**, so the pool maintenance thread leaves it alone and goes back to sleep.

60 seconds later, `sendMessage()` is called again. This time, when the method calls `connectionFactory.createConnection()`, the connection manager discovers that there is a connection, `c1`, available in the free pool for `jms/CF1`. The connection manager takes `c1` out of the free pool, and gives that connection to the application.

The connection is returned to the free pool when `sendMessage()` exits. 120 seconds later, the pool maintenance thread wakes up again, scans the contents of the free pool for `jms/CF1` and discovers `c1`.

Although the connection was only used 120 seconds ago, the pool maintenance thread closes the connection, because the connection has been in existence for a total of 360 seconds, which is longer than the 300 second value you set for the **Aged timeout** property.

How the Minimum connections property affects the pool maintenance thread

Using the "How MDB listener ports use the connection pool" on page 297 example again, assume that you have two MDBs deployed in your application server, each using a different listener port.

Each listener port is configured to use the `jms/CF1` connection factory, which you have configured with the:

- **Unused timeout** property set to 120 seconds
- **Reap time** property set to 180 seconds
- **Minimum connections** property set to 1

Suppose that the first listener is stopped, and its connection `c1` is put into the free pool. 180 seconds later, the pool maintenance thread wakes up, scans the contents of the free pool for `jms/CF1`, and discovers that `c1` has been in the free pool for longer than the value of the **Unused timeout** property for the connection factory.

However, before closing `c1`, the pool maintenance thread looks to see how many connections will remain in the pool if this connection is thrown away. Since `c1` is the only connection in the free connection pool, the connection manager does not close it, because doing so would make the number of connections that remain in the free pool less than the value set for **Minimum connections**.

Now, assume that the second listener is stopped. The free connection pool now contains two free connections - `c1` and `c2`.

180 seconds later, the pool maintenance thread runs again. By this time, `c1` has been in the free connection pool for 360 seconds, and `c2` for 180 seconds.

The pool maintenance thread checks `c1` and discovers that it has been in the pool for longer than the value of the **Unused timeout** property.

The thread then checks to see how many connections are in the free pool, and compares this to the value of the **Minimum connections** property. Since the pool contains two connections, and **Minimum connections** is set to 1, the connection manager closes `c1`.

The maintenance thread now looks at `c2`. This has also been in the free connection pool for longer than the value of the **Unused timeout** property. However, since closing `c2` would leave the free connection pool with less than the set number of Minimum connections in it, the connection manager leaves `c2` alone.

JMS connections and IBM MQ
Information on the use of IBM MQ as the JMS provider.

Using the bindings transport

If a connection factory has been configured to use the bindings transport, every JMS connection establishes a conversation (also known as an **hconn**) with IBM MQ. The conversation uses interprocess communication (or shared memory) to communicate with the queue manager.

Using the client transport

When an IBM MQ messaging provider connection factory has been configured to use the client transport, every connection created from that factory will establish a new conversation (also known as an**hconn**) to IBM MQ.

For connection factories that connect to a queue manager using IBM MQ messaging provider normal mode, it is possible for multiple JMS connections created from the connection factory to share a TCP/IP

connection to IBM MQ. For more information see “Sharing a TCP/IP connection in IBM MQ classes for JMS” on page 305.

To determine the maximum number of client channels used by JMS connections at any one time, add up the value of the *Maximum connections* property for all of the connection factories that point to the same queue manager.

For example, suppose you have two connection factories, `jms/CF1` and `jms/CF2` , that have been configured to connect to the same IBM MQ queue manager using the same IBM MQ channel.

These factories are using the default connection pool properties, which means that *Maximum connections* is set to 10. If all of the connections are being used from both jms/CF1 and jms/CF2 at the same time, there will be 20 conversations between the application server and IBM MQ.

If the connection factory connects to the queue manager using IBM MQ messaging provider normal mode, then the maximum number of TCP/IP connections that can exist between the application server and the queue manager for these connection factories is:

```
20/the value of SHARECNV for the IBM MQ channel
```

If the connection factory is configured to connect using IBM MQ messaging provider migration mode, then the maximum number of TCP/IP connections between the application server and IBM MQ for these connection factories would be 20 (one for each JMS connection in the connection pools for the two factories).

Related concepts

“Using IBM MQ classes for JMS/Jakarta Messaging” on page 79
IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

Object pooling in a Java SE environment

With Java SE (or with another framework such as Spring) the programming models are extremely flexible. Therefore a single pooling strategy does not suit all. You should consider if there is a framework in place that could do any form of pooling, for example, Spring.

Otherwise, application logic could take this up. Ask yourself how complex is the application itself? It is best to understand the application and what it demands from the connectivity to the messaging system. Applications are often written as well within their own wrapper code around the basic JMS API.

Whilst this can be a very sensible approach, and can hide complexity, it is worth keeping in mind that it can introduce problems. For example, a generic `getMessage()` method, that is frequently called, should not just open and close consumers.

Points you should consider:

- How long will the application need access to IBM MQ? All the time, or just occasionally.
- How often will messages be sent? The less frequently, the more a single connection to IBM MQ could be shared.
- A connection broken exception is usually a sign of needing to re-create a pooled connection. What about:
  - Security exceptions or host not available
  - Queue full exceptions
- If a connection broken exception occurs, what should happen to the other free connections in the pool? Should they be closed off and re-created?
- If TLS is being used, for example, how long do you want a single connection to remain open?
- How will a pooled connection identify itself such that a queue manager administrator can spot the connection and track it back.

You should consider all JMS objects for pooling, and pool that object whenever it is possible to do so. The objects include:

- JMS connections
- Session
- Contexts
- Producers and consumers of all different types

When using the client transport, JMS connections, sessions, and contexts, will use sockets when communicating with the IBM MQ queue manager. By pooling these objects, the savings are on the number of incoming IBM MQ connections (`hConns`) to the queue manager and a reduction in the number of channel instances.

Using the bindings transport to the queue manager removes the networking layer entirely. However, many applications use the client transport to provide a more highly available, and workload balanced, configuration.

JMS producers and consumers open destinations on the queue manager. If fewer numbers of queues or topics are opened, and multiple parts of the application are using these objects, pooling these can be useful.

From an IBM MQ perspective, this process saves a sequence of MQOPEN and MQCLOSE operations.

Connections, sessions, and contexts

These objects all encapsulate IBM MQ connection handles to the queue manager, and are generated from a `ConnectionFactory`. You can add logic to an application to constrain the number of connections, and other objects, created from a single connection factory to a specific number.

You can use a simple data structure in the application to contain the connections that are created. The application code that needs to use one of these data structures can *check-out* an object to use.

Take the following factors into account:

- When should connections be removed from the pool? Generally, create an exception listener on the connection. When that listener is called to process an exception, you should re-create the connection, and any sessions created from that connection.
- If a CCDT is in use for workload balancing, the connections could go to different queue managers. This might be applicable for the pooling requirements.

Remember, that the JMS specification states that it is a programming error for multiple threads to be accessing a session or context at the same time. The IBM MQ JMS code does attempt to be rigorous in its handling of threads. However, you should add logic to the application, to ensure that a session or context object is only used by one thread at a time.

Producers and consumers

Each producer and consumer that is created opens a destination on the queue manager. If the same destination is going to be used for a variety of tasks, it makes sense to keep the consumer or producer objects open. Only close the object when all the work is done.

Although opening and closing a destination are short operations, if they are done frequently the time taken can add up.

The scope of these objects is within the session or context they are created from, therefore, they need to be held within that scope. Generally, applications are written such that this is quite straightforward to do.

Monitoring

How will the applications monitor their object pools? The answer to this is largely determined by the complexity of the solution of pooling implemented.

If you consider a JavaEE pooling implementation, there are a large number of options, including the:

- Current size of the pools
- Time objects have spent in them
- Cleaning of the pools
- Refreshing of the connections

You should also consider how a single re-used session appears on the queue manager. There are connection factory properties to identify the application (such as appName) that could be useful.

"Using IBM MQ classes for JMS/Jakarta Messaging" on page 79
IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

Sharing a TCP/IP connection in IBM MQ classes for JMS
Multiple instances of an MQI channel can be made to share a single TCP/IP connection.

Applications that are running inside the same Java runtime environment, and that use the IBM MQ classes for JMS or the IBM MQ resource adapter to connect to a queue manager by using the CLIENT transport, can be made to share a channel instance.

If a channel is defined with the **SHARECNV** parameter set to a value greater than 1, then that number of conversations can share a channel instance. To enable a connection factory or an activation specification to use this function, set the **SHARECONVALLOWED** property to YES.

Every JMS connection and JMS session that is created by a JMS application creates its own conversation with the queue manager.

When an activation specification starts up, the IBM MQ resource adapter starts a conversation with the queue manager for the activation specification to use. Every server session in the server session pool that is associated with the activation specification also starts a conversation with the queue manager.

The **SHARECNV** attribute is a best effort approach to connection sharing. Therefore, when a **SHARECNV** value greater than 0 is used with the IBM MQ classes for JMS, it is not guaranteed that a new connection request always shares an already established connection.

How the TCP/IP connections are shared

There are two strategies available for sharing TCP/IP connections:

The GLOBAL strategy
This strategy is the default strategy for sharing TCP/IP connections. Any JMS connection or session can use a conversation on any suitable TCP/IP connection. Suitability is determined by such factors as host address, port number, user ID and password, and TLS/SSL parameters.

This approach for sharing TCP/IP connections minimizes the number of channel instances that are in use, but at the cost of contention for access to a global pool of TCP/IP connections.

The CONNECTION strategy
With this strategy, the channel instances are only shared between related JMS objects. Specifically, when a JMS connection is created, a channel instance is created for it, and additional conversations on that channel instance are only available to JMS sessions that are created by that JMS connection.

If more conversations are created than the SHARECNV attribute specifies, a new channel instance is created which can be used only by JMS sessions that are created by the original JMS connection.

This approach to sharing channel instances reduces contention for conversations, at the expense of potentially requiring significantly more channel instances.

Explicitly specifying a channel instance sharing strategy

V 9.4.0

By default, the GLOBAL strategy is used if applications are not reconnectable. Reconnectable applications always use the CONNECTION strategy.

For applications that use IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging, the CONNECTION strategy can be enabled for non-reconnectable applications on an application-wide basis. You can enable the CONNECTION strategy by setting the system property `com.ibm.mq.jms.channel.sharing` to the value `CONNECTION`. This value is not case-sensitive, and any value other than `CONNECTION` is ignored.

You can set the system property `com.ibm.mq.jms.channel.sharing` in one of the following ways:

- Set the property as part of the JVM initialization by using the "-D" command line option:

```
-Dcom.ibm.mq.jms.channel.sharing=CONNECTION
```

- Set the property before any use of the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging by using `System.setProperty()`

Calculating the number of channel instances for the GLOBAL sharing strategy

Use the following formulae to determine the maximum number of channel instances that are created by an application:

Activation specifications

Number of channel instances = (*maxPoolDepth_value* + 1) / *SHARECNV_value*

Where *maxPoolDepth_value* is the value of the **maxPoolDepth** property and *SHARECNV_value* is the value of the **SHARECNV** property on the channel that is used by the activation specification.

Other JMS applications

Number of channel instances = (*jms_connections* + *jms_sessions*) / *SHARECNV_value*

Where *jms_connections* is the number of connections that a created by the application, *jms_sessions* is the number of JMS sessions that are created by the application, and *SHARECNV_value* is the value of the **SHARECNV** property on the channel that is used by the activation specification.

Calculating the number of channel instances for the CONNECTION sharing strategy

The number of channel instances depends on the distribution of JMS sessions among the JMS connections in the application.

Allow one conversation for the JMS connection and one conversation for each JMS session under that JMS connection, then divide by the **SHARECNV** value, rounding up. This calculation gives the channel instances that are needed by that JMS connection.

The same principle can be applied to activation specifications. Regard the activation specification as a JMS connection and the **maxPoolDepth** property as the number of JMS sessions.

Examples

The following examples show how to use the formulae to calculate the number of channel instances that are created on a queue manager by applications by using either the IBM MQ classes for JMS or the IBM MQ resource adapter.

JMS application example

A JMS application connection connects to a queue manager by using the CLIENT transport, and creates a JMS connection and three JMS sessions. The channel that the application is using to connect to the queue manager has the **SHARECNV** property set to the value of 10. When the application is running, there are four conversations between the application and the queue manager and one channel instance. The four conversations all share the channel instance.

Activation specification example

An activation specification connects to a queue manager by using the CLIENT transport. The activation specification is configured with the **maxPoolDepth** property set to 10. The channel that the activation specification is configured to use has the **SHARECNV** property set to 10. When the activation specification is running, and processing 10 messages concurrently, the number of conversations between the activation specification and the queue manager is 11 (10 conversations for the server sessions, and one for the activation specification). The number of channel instances that are used by the activation specification is 2.

Activation specification example

An activation specification connects to a queue manager by using the CLIENT transport. The activation specification is configured with the **maxPoolDepth** property set to 5. The channel that the activation specification is configured to use has the **SHARECNV** property set to 0. When the activation specification is running, and processing 5 messages concurrently, the number of conversations between the activation specification and the queue manager is 6 (five conversations for the server sessions, and one for the activation specification). The number of channel instances that are used by the activation specification is 6 because the **SHARECNV** property on the channel is set to 0, every conversation uses its own channel instance.

Related tasks

"Determining the number of TCP/IP connections that are created from WebSphere Application Server to IBM MQ" on page 484

Using the sharing conversations feature, multiple conversations can share MQI channel instances, this is also known as a TCP/IP connection.

Specifying a range of ports for client connections in IBM MQ classes for JMS

Use the LOCALADDRESS property to specify a range of ports that your application can bind to.

When an IBM MQ classes for JMS application attempts to connect to an IBM MQ queue manager in client mode, a firewall might allow only those connections that originate from specified ports or a range of ports. In this situation, you can use the LOCALADDRESS property of a ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory object to specify a port, or a range of ports, that the application can bind to.

You can set the LOCALADDRESS property by using the IBM MQ JMS administration tool, or by calling the setLocalAddress() method in a JMS application. Here is an example of setting the property from within an application:

```
mqConnectionFactory.setLocalAddress("192.0.2.0(2000,3000)");
```

When the application connects to a queue manager subsequently, the application binds to a local IP address and port number in the range 192.0.2.0(2000) to 192.0.2.0(3000).

In a system with more than one network interface, you can also use the LOCALADDRESS property to specify which network interface must be used for a connection.

For a real-time connection to a broker, the LOCALADDRESS property is relevant only when multicast is used. In this case, you can use the property to specify which local network interface must be used for a connection, but the value of the property must not contain a port number, or a range of port numbers.

Connection errors might occur if you restrict the range of ports. If an error occurs, a JMSException is thrown with an embedded MQException that contains the IBM MQ reason code MQRC_Q_MGR_NOT_AVAILABLE and the following message:

```
Socket connection attempt refused due to LOCAL_ADDRESS_PROPERTY restrictions
```

An error might occur if all the ports in the specified range are in use, or if the specified IP address, host name, or port number is not valid (a negative port number, for example).

Because IBM MQ classes for JMS might create connections other than those required by an application, always consider specifying a range of ports. In general, every session created by an application requires

one port and IBM MQ classes for JMS might require three or four additional ports. If a connection error does occur, increase the range of ports.

Connection pooling, which is used by default in IBM MQ classes for JMS, might have an effect on the speed at which ports can be reused. As a result, a connection error might occur while ports are being freed.

Channel compression in IBM MQ classes for JMS
An IBM MQ classes for JMS application can use IBM MQ facilities to compress a message header or data.

Compressing the data that flows on an IBM MQ channel can improve the performance of the channel and reduce network traffic. Using function supplied with IBM MQ, you can compress the data that flows on message channels and MQI channels. On either type of channel, you can compress header data and message data independently of each other. By default, no data is compressed on a channel.

An IBM MQ classes for JMS application specifies the techniques that can be used for compressing header or message data on a connection by creating a java.util.Collection object. Each compression technique is an Integer object in the collection, and the order in which the application adds the compression techniques to the collection is the order in which the compression techniques are negotiated with the queue manager when the application creates the connection. The application can then pass the collection to a ConnectionFactory object by calling the setHdrCompList() method, for header data, or the setMsgCompList() method, for message data. When the application is ready, it can create the connection.

The following code fragments illustrate the approach described. The first code fragment shows you how to implement header data compression:

```
Collection headerComp = new Vector();
headerComp.add(new Integer(WMQConstants.WMQ_COMPHDR_SYSTEM));
.
.
.
((MQConnectionFactory) cf).setHdrCompList(headerComp);
.
.
.
connection = cf.createConnection();
```

The second code fragment shows you how to implement message data compression:

```
Collection msgComp = new Vector();
msgComp.add(new Integer(WMQConstants.WMQ_COMPMSG_RLE));
msgComp.add(new Integer(WMQConstants.WMQ_COMPMSG_ZLIBHIGH));
.
.
.
((MQConnectionFactory) cf).setMsgCompList(msgComp);
.
.
.
connection = cf.createConnection();
```

In the second example, the compression techniques are negotiated in the order RLE, then ZLIBHIGH, when the connection is created. The compression technique that is selected cannot be changed during the lifetime of the Connection object. To use compression on a connection, the setHdrCompList() and the setMsgCompList() methods must be called before creating the Connection object.

Putting messages asynchronously in IBM MQ classes for JMS
Normally, when an application sends messages to a destination, the application has to wait for the queue manager to confirm that it has processed the request. You can improve messaging performance in some circumstances by choosing instead to put messages asynchronously. When an application puts a message asynchronously, the queue manager does not return the success or failure of each call, but you can instead check for errors periodically.

Whether a destination returns control to the application, without determining whether the queue manager has received the message safely, depends upon the following properties:

The JMS destination property PUTASYNCALLOWED (short name - PAALD).
PUTASYNCALLOWED controls whether JMS applications can put messages asynchronously, if this option is allowed by the underlying queue or topic that the JMS destination represents.

The IBM MQ queue or topic property DEFPRESP (Default put response type).
DEFPRESP specifies whether applications that put messages to the queue, or publish messages to the topic, can use asynchronous put functionality.

The following table shows the possible values for the PUTASYNCALLOWED and DEFPRESP properties, and the combinations of values used to enable asynchronous put functionality:

Table 46. How the PUTASYNCALLOWED and DEFPRESP properties combine to determine if messages are put to a destination asynchronously.

IBM MQ queue property	PUTASYNCALLOWED = NO	PUTASYNCALLOWED = YES	Asynchronous put functionality enabled
DEFPRESP=SYNC	Asynchronous put functionality not enabled	Asynchronous put functionality enabled	PUTASYNCALLOWED = AS_DEST or AS_Q_DEF or AS_T_DEF
DEFPRESP=ASYNC	Asynchronous put functionality not enabled	Asynchronous put functionality enabled	PUTASYNCALLOWED = AS_DEST or AS_Q_DEF or AS_T_DEF

You can change the behavior by specifying the IBM MQ-JMS Destination property to say "NO" or "YES" as shown in the table, but it can also be overridden for the entire Java Virtual Machine using the JVM **SystemProperty** and value:

```
com.ibm.mq.cfg.Channels.Put1DefaultAlwaysSync=Y
```

For messages sent in a transacted session, the application ultimately determines whether the queue manager has received the messages safely when it calls `commit()`.

If an application sends persistent messages within a transacted session, and one or more of the messages are not received safely, the transaction fails to commit and produces an exception. However, if an application sends nonpersistent messages within a transacted session, and one or more of the messages are not received safely, the transaction commits successfully. The application does not receive any feedback that the nonpersistent messages did not arrive safely.

For nonpersistent messages sent in a session that is not transacted, the SENDCHECKCOUNT property of the *ConnectionFactory* object specifies how many messages are to be sent, before IBM MQ classes for JMS checks that the queue manager has received the messages safely.

If a check discovers that one or more messages were not received safely, and the application has registered an exception listener with the connection, IBM MQ classes for JMS calls the `onException()` method of the exception listener to pass a JMS exception to the application.

The JMS exception has an error code of JMSWMQ0028 and this code displays the following message:

```
At least one asynchronous put message failed or gave a warning.
```

The JMS exception also has a linked exception that provides more details. The default value of the SENDCHECKCOUNT property is zero, which means that no such checks are made.

This optimization is of most benefit to an application that connects to a queue manager in client mode, and needs to send a sequence of messages in rapid succession, but does not require immediate feedback from the queue manager for each message sent. However, an application can still use this optimization even if it connects to a queue manager in bindings mode, but the expected performance benefit is not as great.

Note: If you are using an unidentified **MessageProducer** to send a message under a transaction, then by default the messages are put to the queue using the asynchronous put mechanism.

This can occur because the JMS API allows the **MessageProducer** to be created without specifying a Destination, using the syntax:

```
javax.jms.MessageProducer messageProducer =  javax.jms.Session.createProducer(null);
  messageProducer.send(Destination destination, Message message, int deliveryMode, int priority, long
timeToLive);
```

In this scenario, the JMS Destination is provided when the message is sent rather than ahead of time when the **MessageProducer** is constructed. In terms of the IBM MQ API, this results in an MQPUT1 being issued to put the message to the queue .

If you do this under an IBM MQ syncpoint, which means (in JMS terminology) putting the message under a transaction, either using a transacted JMS Session or through the use of an XASession the IBM MQ classes for JMS API switches to using asynchronous put.

Using read ahead with IBM MQ classes for JMS
The read ahead functionality that is provided by IBM MQ allows non-persistent messages that are received outside of a transaction to be sent to the IBM MQ classes for JMS before an application requests them. The IBM MQ classes for JMS store the messages in an internal buffer, and pass the messages to the application when the application asks for them.

IBM MQ classes for JMS applications that use MessageConsumers or MessageListeners to receive messages from a destination outside of a transaction can use the read ahead functionality. Using read ahead allows applications that use these objects to benefit from improved performance when they receive messages.

Whether an application that uses MessageConsumers or MessageListeners can use read ahead depends upon the following properties:

The JMS destination property READAHEADALLOWED (short name - RAALD).
READAHEADALLOWED controls whether JMS applications can use read ahead when getting or browsing non-persistent messages outside of a transaction, if the underlying queue or topic that the JMS destination represents, allows this option.

The IBM MQ queue or topic property DEFREADA (Default read ahead).
DEFREADA specifies whether applications that are receiving or browsing non-persistent messages outside of a transaction can use read ahead.

The following table shows the possible values for the READAHEADALLOWED and DEFREADA properties, and the combinations of values used to enable the read ahead functionality:

Table 47. How the READAHEADALLOWED and DEFREADA properties combine to determine if read ahead is used when receiving or browsing non-persistent messages outside of a transaction.

IBM MQ queue property	READAHEADALLOWED = YES	READAHEADALLOWED = NO	AS_DEST or AS_Q_DEF or AS_T_DEF
DEFREADA = NO	Read ahead functionality enabled	Read ahead functionality not enabled	Read ahead functionality not enabled
DEFREADA = YES	Read ahead functionality enabled	Read ahead functionality not enabled	Read ahead functionality enabled
DEFREADA = DISABLED	Read ahead functionality not enabled	Read ahead functionality not enabled	Read ahead functionality not enabled

If the read ahead functionality is enabled, when a MessageConsumer or MessageListener is created by an application, the IBM MQ classes for JMS create an internal buffer for the destination that the MessageConsumer or MessageListener is monitoring. There is one internal buffer for each MessageConsumer or MessageListener. The queue manager starts sending non-persistent messages to the IBM MQ classes for JMS when the application calls one of the following methods:

- MessageConsumer.receive()

- `MessageConsumer.receive(long timeout)`
- `MessageConsumer.receiveNoWait()`
- `Session.setMessageListener(MessageListener listener)`

The IBM MQ classes for JMS automatically returns the first message back to the application, by the method call that the application has made. The other non-persistent messages are stored by the IBM MQ classes for JMS in the internal buffer that was created for the destination. When the application requests the next message to process, the IBM MQ classes for JMS will return the next message in the internal buffer.

The IBM MQ classes for JMS requests more non-persistent messages from the queue manager when the internal buffer is empty.

The internal buffer that is used by the IBM MQ classes for JMS is deleted when an application closes a `MessageConsumer`, or the JMS Session that a `MessageListener` is associated with.

For `MessageConsumers`, any unprocessed messages in the internal buffer is lost.

When using `MessageListeners`, what happens to the messages in the internal buffer depends upon the JMS destination property READAHEADCLOSEPOLICY (short name - RACP). The default value of the property is DELIVER_ALL, which means that the JMS session that was used to create the `MessageListener` is not closed until all of the messages in the internal buffer are delivered to the application. If the property is set to DELIVER_CURRENT, then the JMS session will be closed after the current message has been processed by the application and all of the remaining messages in the internal buffer are discarded.

Retained publications in IBM MQ classes for JMS
An IBM MQ classes for JMS client can be configured to use retained publications.

A publisher can specify that a copy of a publication must be retained so that it can be sent to future subscribers who register an interest in the topic. This is done in IBM MQ classes for JMS by setting the integer property JMS_IBM_RETAIN to the value 1. Constants have been defined for these values in the com.ibm.msg.client.jms.JmsConstants interface. For example, if you have created a message *msg*, to set it as a retained publication use the following code:

```
// set as a retained publication
msg.setIntProperty(JmsConstants.JMS_IBM_RETAIN, JmsConstants.RETAIN_PUBLICATION);
```

You can now send the message as normal. JMS_IBM_RETAIN can also be queried in a received message. It is therefore possible to query whether a received message is a retained publication.

XA support in IBM MQ classes for JMS

JMS supports XA-compliant transactions in bindings and client modes with a supported transaction manager within a JEE container.

If you require XA functionality in an application server environment, you must configure your application appropriately. Refer to your application server's own documentation for information about how to configure applications to use distributed transactions.

An IBM MQ queue manager cannot act as a transaction manager for JMS.

Delivery delay for JMS messages

For JMS 2.0 or later, you can specify a delivery delay when sending a message. The queue manager does not deliver the message until after the specified delivery delay has elapsed.

An application can specify a delivery delay in milliseconds, when it sends a message, by using either `MessageProducer.setDeliveryDelay(long deliveryDelay)` or `JMSProducer.setDeliveryDelay(long deliveryDelay)`. This value is added to the time at which the message is sent and gives the earliest time at which any other application can get that message.

Delivery delay is implemented by using a single internal staging queue. Messages that have a nonzero delivery delay are placed on this queue with a header that indicates the delivery delay and information

about the target queue. A component of the queue manager that is called the delivery delay processor monitors the messages on the staging queue. When a message's delivery delay completes, the message is taken off the staging queue and placed on the target queue.

Messaging clients

The IBM MQ implementation of delivery delay is only available for use when you are using the JMS client. The following restrictions apply if you are using delivery delay with IBM MQ. These restrictions apply equally to `MessageProducers` and `JMSProducers`, but `JMSRuntimeExceptions` are thrown in the case of `JMSProducers`.

- Any attempt to call `MessageProducer.setDeliveryDelay` with a nonzero value when connected to a queue manager earlier than IBM MQ 8.0, results in a `JMSException` with a MQRC_FUNCTION_NOT_SUPPORTED message.
- Delivery delay is not supported for clustered destinations that have a **DEFBIND** value other than MQBND_BIND_NOT_FIXED. If a `MessageProducer` has a nonzero delivery delay set and an attempt is made to send to a destination that does not meet this requirement, then the call results in a `JMSException` with an MQRC_OPTIONS_ERROR message.
- Any attempt to set a time to live value that is less than a previously specified nonzero delivery delay, or vice versa, results in a `JMSException` with a MQRC_EXPIRY_ERROR message. This checking is done on calling `setTimeToLive` or `setDeliveryDelay` or send methods, depending on the exact set of operations chosen.
- Use of retained publications and delivery delay is not supported. Attempting to publish a message with a delivery delay if that message has been marked as retained by using `msg.setIntProperty(JmsConstants.JMS_IBM_RETAIN, JmsConstants.RETAIN_PUBLICATION)` results in a `JMSException` with a MQRC_OPTIONS_ERROR message.
- Delivery delay and message grouping is not supported and any attempt to use this combination results in a `JMSException` with a MQRC_OPTIONS_ERROR message.

Any failure to send a message with delivery delay results in the client throwing a `JMSException` with a suitable error message, for example queue full. In some situations, the error message might apply to the target destination, or the staging queue, or both.

Note: IBM MQ allows applications that put a message in a unit of work to get the same message again even though the unit of work has not committed. This technique does not work with delivery delay as the message is not placed on the staging queue until the unit of work is committed, and as a result will not have been sent to the target destination.

Authorization

IBM MQ carries out authorization checks on the original target destination when the application sends a message with a nonzero delivery delay. If the application is not authorized, then the send fails. When the queue manager detects that a message's delivery delay is complete, it opens the target queue. No authorization checks are carried out at this point.

SYSTEM.DDELAY.LOCAL.QUEUE

A system queue, SYSTEM.DDELAY.LOCAL.QUEUE, is used to implement delivery delay.

- Multi On Multiplatforms, SYSTEM.DDELAY.LOCAL.QUEUE exists by default. The system queue must be altered so that its MAXMSGL and MAXDEPTH attributes are sufficient for the expected load.
- z/OS On IBM MQ for z/OS, SYSTEM.DDELAY.LOCAL.QUEUE is used as a staging queue for messages that are sent with delivery delay to both local and shared queues. On z/OS, the queue must be created and be defined so that its MAXMSGL and MAXDEPTH attributes are sufficient for the expected load.

When this queue is created, it must be secured so that as few users as possible have access to it. Access to the queue must be for maintenance and monitoring purposes only.

When a message is sent by a JMS application with a nonzero delivery delay, it is put to this queue with a new message ID. The original message ID is placed in the correlation ID of the message. This correlation ID allows an application to retrieve a message from the staging queue when required, for example if a large delivery delay was used by mistake.

Considerations for z/OS

z/OS

If your system is running on z/OS, there are additional considerations to take into account if you want to use delivery delay.

If delivery delay is to be used, the system queue SYSTEM.DDELAY.LOCAL.QUEUE must be defined. It must be defined with a storage class that is sufficient for its expected load, and with INDXTYPE(NONE), and MSGDLVSQ(FIFO) specified. A sample definition of the system queue is provided, commented out, in the CSQ4INSG JCL.

Shared queues

Delivery delay is supported for sending messages to shared queues. However, there is only a single, private staging queue that is used regardless of whether the target queue is shared or not. The queue manager that owns that private queue must be running to send the delayed message to its target shared queue when the delay completes.

Note: If a non-persistent message is put with a delivery delay to a shared queue, and the queue manager that owns the staging queue shuts down, the original message is lost. As a result non-persistent messages sent with delivery delay to a shared queue are more likely to be lost than non-persistent messages sent without delivery delay to a shared queue.

Target destination resolution

If the message is sent to a queue, resolution is driven twice; once by the JMS application and once by the queue manager when it takes the message off the staging queue and sends it to the target queue.

Target subscriptions for publications are matched when the JMS application calls the send method.

If a message is sent with persistence or priority according to the queue definition, then the value is set on the first resolution and not the second.

Expiry interval

Delivery delay preserves the behavior of the expiry property, **`MQMD.Expiry`**. For example, if a message was put from a JMS application with an expiry interval of 20,000 ms and a delivery delay of 5,000 ms, and got after an elapsed time of 10,000 ms, then the value of the MQMD.expiry field might be approximately 50 tenths of a second. This value indicates that 15 seconds has elapsed from the time the message was put, to the time when it was got.

If a message expires while on the staging queue and one of the MQRO_EXPIRATION_* options is set, then the report generated is for the original message as sent by the application, the header used to contain the delivery delay information is removed.

Stopping and starting the delivery delay processor

z/OS On z/OS, the delivery delay processor is integrated into the queue manager MSTR address space. When the queue manager starts, the delivery delay processor also starts. If the staging queue is available, it opens the queue and waits for messages to arrive on it to be processed. If the staging queue has not been defined, or is disabled for gets, or another error occurs, the delivery delay processor shuts down. If the staging queue is later defined, or altered to be get enabled, the delivery delay processor

restarts. If the delivery delay processor shuts down for any other reason, it can be restarted by altering the **PUT** attribute of the staging queue from ENABLED to DISABLED and back to ENABLED again. Should you need to stop the delivery delay processor for any reason, set the PUT attribute of the staging queue to DISABLED.

On Multiplatforms, the delay processor starts with the queue manager, and is automatically restarted in the event of a recoverable failure.

Failure to put to target queue

If a delayed message cannot be put to the target queue once its delay completes, the message is dealt with as indicated in its report options: it is either discarded or sent to the dead letter queue. If this action fails, then an attempt is made to put the message later. If the action is successful an exception report is generated and sent to the specified queue, if the report is requested. If the report message could not be sent, the report message is sent to the dead letter queue. If sending the report to the dead letter queue fails and the message is persistent, all changes are discarded and the original message rolled back and redelivered later. If the message is non-persistent the report message is discarded, but other changes are committed. If a delayed publication cannot be delivered because a subscriber has unsubscribed, or in the case of a non-durable subscriber, because it has disconnected, then the message is discarded silently. Report messages are still generated as described earlier.

If a delayed publication cannot be delivered to a subscriber and is instead put to the dead letter queue, and the put to the dead letter queue fails then the message is discarded.

To reduce the likelihood of the put to the target queue failing after the delivery delay has completed, the queue manager performs some basic checks when the JMS client sends a message with a nonzero delivery delay. These checks include whether the queue is put disabled, if the message is bigger than the maximum message length allowed, and if the queue is full.

Publish/subscribe

Matching of a publication to available subscriptions occurs when the JMS application sends a message with a nonzero delivery delay. A message for each matching subscriber is put to the SYSTEM.DDELAY.LOCAL.QUEUE queue, where it is kept until the delivery delay completes. If one of those subscribers is a proxy subscription for another queue manager, then the fan-out on that queue manager occurs after the delivery delay is complete. This might result in subscribers on the other queue manager receiving publications that were originally published before they subscribed. This is a deviation from the JMS 2.0 or later specification.

Delivery delay with publish/subscribe is only supported if the target topic is configured with (N)PMSGDLV = ALLAVAIL. An attempt to use any other values results in a MQRC_PUBLICATION_FAILURE error. If the delivery delay processor fails while it is putting the message to the target queue, the result is as described in the "Failure to put to target queue" section.

Report messages

All report options are supported and actioned by the delivery processor, other than the following options that are ignored, but passed through on the message when it is sent to the target queue:

- MQRO_COA*
- MQRO_COD*
- MQRO_PAN/MQRO_NAN
- MQRO_ACTIVITY

Cloned and shared subscriptions

There are two methods for giving multiple consumers access to the same subscription. These two methods are by using cloned subscriptions, or by using shared subscriptions.

Cloned subscriptions

Cloned subscription is an IBM MQ extension. Cloned subscriptions enable multiple consumers in different Java virtual machines (JVMs) concurrent access to the subscription. This behavior can be used by setting the **CLONESUPP** property to `Enabled` on a ConnectionFactory object. By default **CLONESUPP** is `Disabled`. Cloned subscriptions can be enabled only on durable subscriptions. If **CLONESUPP** is enabled, each subsequent connection that is made by using this ConnectionFactory is cloned.

A durable subscription can be considered cloned if one or more consumers are created to receive messages from that subscription, that is, they were created specifying the same subscription name. This can be done only if the connection under which the consumers were created has **CLONESUPP** set to `Enabled` on the MQConnectionFactory. When a message is published on the subscription's topic, a copy of that message is sent to the subscription. The message is available to any of the consumers, but only one receives it.

Note: Enabling cloned subscriptions extends the JMS specification.

Shared subscriptions

Shared subscriptions, which were introduced by the JMS 2.0 specification, allow messages from a topic subscription to be shared among multiple consumers. Each message from the subscription is delivered to only one of the consumers on that subscription. Shared subscriptions are enabled by the relevant call to the JMS 2.0 or later API.

The APIs can be called in either of the following ways:

- From a Java SE application (or Java EE Client Container).
- From a servlet or the implementation of an MDB.

The JMS 2.0 or later specification does not define any standard way of driving an MDB from a shared subscription, so IBM MQ provides the **sharedSubscription** activation specification property for this purpose. For more information about this property, see “Configuring the resource adapter for inbound communication” on page 436 and “Examples of how to define the sharedSubscription property” on page 452.

If a shared subscription is enabled, then it cannot be unshared.

Shared subscriptions can be created as either durable or non-durable subscriptions. There is no requirement to separately create any objects on the queue manager side beyond the normal JMS configuration. Any objects that are required are created dynamically.

Deciding between shared or cloned subscriptions

When you are deciding whether to use shared or cloned subscriptions, consider the benefits of both. Where possible, use shared subscriptions as it is specification-defined behavior, rather than an IBM MQ specific extension.

The following table contains some of the points to consider when you are deciding between shared and cloned subscriptions:

Table 48. Comparison of considerations for shared subscriptions and cloned subscriptions

Shared Subscriptions	Cloned Subscriptions
Shared subscriptions are a standard part of the JMS 2.0 or later specification.	Cloned subscriptions are an IBM MQ specific extension.

Table 48. Comparison of considerations for shared subscriptions and cloned subscriptions (continued)	
Shared Subscriptions	**Cloned Subscriptions**
Shared subscriptions are created by using explicit API method calls.	Cloned subscriptions are controlled administratively at the ConnectionFactory level.
Shared subscriptions can be durable or nondurable.	Cloned subscriptions can only be durable.
Shared subscriptions are explicitly created on an individual subscription basis.	Cloned subscriptions are used for any durable subscription under a connection for which the function is enabled.
If a subscription is created as shared it cannot later be changed to unshared, or vice versa.	A subscription can be changed from cloned to uncloned each time it is reopened if the **CLONESUPP** property of the owning connection has changed.

Related concepts

Subscribers and subscriptions

Subscription durability

Related tasks

Using JMS 2.0 shared subscriptions

Related reference

"Examples of how to define the sharedSubscription property" on page 452

You can define the sharedSubscription property of an activation specification within a WebSphere Liberty `server.xml` file. Alternatively, you can define the property within a message driven bean (MDB) using annotations.

CLONESUPP

SupportMQExtensions property

The JMS 2.0 specification introduced changes to the way certain behaviors work. IBM MQ 8.0 and later includes the property **com.ibm.mq.jms.SupportMQExtensions**, which can be set to TRUE to revert these changed behaviors back to previous implementations.

JMS 3.0 The **com.ibm.mq.jakarta.jms.SupportMQExtensions** property (Jakarta Messaging 3.0) is supported by the IBM MQ classes for Jakarta Messaging, which are available in `com.ibm.mq.jakarta.client.jar`.

JMS 2.0 The **com.ibm.mq.jms.SupportMQExtensions** (JMS 2.0) property is supported by the IBM MQ classes for JMS, which are available in `com.ibm.mq.allclient.jar` or `com.ibm.mqjms.jar`.

Three areas of functionality are reverted by setting **SupportMQExtensions** to `True`:

Message priority
: Messages can be assigned a priority of 0 - 9. Before JMS 2.0, messages could also use the value -1, indicating that a queue's default priority is used. JMS 2.0 and later do not allow a message priority of -1 to be set. Turning on **SupportMQExtensions** allows the value of -1 to be used.

Client id
: The JMS 2.0 or later specification requires that non-null client IDs are checked for uniqueness when they make a connection. Turning on **SupportMQExtensions** means that this requirement is disregarded, and that a client ID can be reused.

NoLocal
: The JMS 2.0 or later specification requires that when this constant is turned on, a consumer cannot receive messages that are published by the same client ID. Before JMS 2.0, this attribute was set on

a subscriber to prevent it receiving messages that are published by its own connection. Turning on **SupportMQExtensions** reverts this behavior to its previous implementation.

This property can be set as follows:

```
java -Dcom.ibm.mq.jms.SupportMQExtensions=true
```

This property can be set either as a standard JVM System property on the **java** command or contained within the IBM MQ classes for JMS configuration file.

Related concepts

"The IBM MQ classes for JMS/Jakarta Messaging configuration file" on page 94
An IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging configuration files specify properties that are used to configure IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging.

Related reference

"Properties used to configure JMS client behavior" on page 101
Use these properties to configure the behavior of the JMS client.

Using shared subscriptions in JMS applications

With shared subscriptions, a single subscription is shared among multiple consumers, with only one of the consumers receiving a publication at any point in time.

Shared subscriptions are available from JMS 2.0 onwards. When you are developing a JMS application for IBM MQ 8.0 or later, you might therefore need to consider the impact of this functionality on your queue manager.

The idea behind shared subscriptions is to share the load among multiple consumers. A durable subscription can also be shared among multiple consumers.

For example, assume there is a :

- Subscription SUB, subscribing to a topic FIFA2014/UPDATES to receive football match updates, being shared by three consumers C1, C2, and C3
- Producer P1 publishing on the FIFA2014/UPDATES topic

When a publication is made on FIFA2014/UPDATES, the publication will be received by only one of the three consumers (C1, or C2, or C3) but not all.

The following sample demonstrates the usage of shared subscriptions, and also demonstrates the usage of the additional API in JMS 2.0, `Message.receiveBody()`, to retrieve only the message body.

The sample creates three subscriber threads, which create a shared subscription to the FIFA2014/UPDATES topic, and one publisher thread.

JMS 3.0

```
package mqv91Samples;

import jakarta.jms.JMSException;

import com.ibm.msg.client.jms.JmsConnectionFactory;
import com.ibm.msg.client.jms.JmsFactoryFactory;
import com.ibm.msg.client.wmq.WMQConstants;

import jakarta.jms.JMSContext;
import jakarta.jms.Topic;
import jakarta.jms.Queue;
import jakarta.jms.JMSConsumer;
import jakarta.jms.Message;
import jakarta.jms.JMSProducer;

/*
* Implements both Subscriber and Publisher
*/
class SharedNonDurableSubscriberAndPublisher implements Runnable {
   private Thread t;
   private String threadName;

   SharedNonDurableSubscriberAndPublisher( String name){
```

```
        threadName = name;
        System.out.println("Creating Thread:" +  threadName );
    }

    /*
     * Demonstrates shared non-durable subscription in JMS 2.0 and later
     */
     private void sharedNonDurableSubscriptionDemo(){
         JmsConnectionFactory cf = null;
         JMSContext msgContext = null;

         try {
             // Create Factory for WMQ JMS provider
             JmsFactoryFactory ff = JmsFactoryFactory.getInstance(WMQConstants.WMQ_PROVIDER);
             // Create connection factory
             cf = ff.createConnectionFactory();
             // Set MQ properties
             cf.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "QM3");
             cf.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE, WMQConstants.WMQ_CM_BINDINGS);
             // Create message context
             msgContext = cf.createContext();

             // Create a topic destination
             Topic fifaScores = msgContext.createTopic("/FIFA2014/UPDATES");

             // Create a consumer. Subscription name specified, required for sharing of subscription.
             JMSConsumer msgCons = msgContext.createSharedConsumer(fifaScores, "FIFA2014SUBID");

             // Loop around to receive publications
             while(true){

                 String msgBody=null;

                 // Use JMS 2.0 and later receiveBody method as we are interested in message body only.
                 msgBody = msgCons.receiveBody(String.class);

                 if(msgBody != null){
                     System.out.println(threadName + " : " + msgBody);
                 }
             }
         }catch(JMSException jmsEx){
             System.out.println(jmsEx);
         }
     }
```

JMS 2.0

```
package mqv91Samples;

import javax.jms.JMSException;

import com.ibm.msg.client.jms.JmsConnectionFactory;
import com.ibm.msg.client.jms.JmsFactoryFactory;
import com.ibm.msg.client.wmq.WMQConstants;

import javax.jms.JMSContext;
import javax.jms.Topic;
import javax.jms.Queue;
import javax.jms.JMSConsumer;
import javax.jms.Message;
import javax.jms.JMSProducer;

/*
* Implements both Subscriber and Publisher
*/
class SharedNonDurableSubscriberAndPublisher implements Runnable {
   private Thread t;
   private String threadName;

   SharedNonDurableSubscriberAndPublisher( String name){
       threadName = name;
       System.out.println("Creating Thread:" +  threadName );
   }

   /*
    * Demonstrates shared non-durable subscription in JMS 2.0 and later
    */
    private void sharedNonDurableSubscriptionDemo(){
        JmsConnectionFactory cf = null;
        JMSContext msgContext = null;
```

```
        try {
            // Create Factory for WMQ JMS provider
            JmsFactoryFactory ff = JmsFactoryFactory.getInstance(WMQConstants.WMQ_PROVIDER);
            // Create connection factory
            cf = ff.createConnectionFactory();
            // Set MQ properties
            cf.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "QM3");
            cf.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE, WMQConstants.WMQ_CM_BINDINGS);
            // Create message context
            msgContext = cf.createContext();

            // Create a topic destination
            Topic fifaScores = msgContext.createTopic("/FIFA2014/UPDATES");

            // Create a consumer. Subscription name specified, required for sharing of subscription.
            JMSConsumer msgCons = msgContext.createSharedConsumer(fifaScores, "FIFA2014SUBID");

            // Loop around to receive publications
            while(true){

                String msgBody=null;

                // Use JMS 2.0 and later receiveBody method as we are interested in message body only.
                msgBody = msgCons.receiveBody(String.class);

                if(msgBody != null){
                    System.out.println(threadName + " : " + msgBody);
                }
            }
        }catch(JMSException jmsEx){
            System.out.println(jmsEx);
        }
    }
```

```
 /*
        * Publisher publishes match updates like current attendance in the stadium, goal score and ball
possession by teams.
        */
        private void matchUpdatePublisher(){
            JmsConnectionFactory cf = null;
            JMSContext msgContext = null;
            int nederlandsGoals = 0;
            int chileGoals = 0;
            int stadiumAttendence = 23231;
            int switchIndex = 0;
            String msgBody = "";
            int nederlandsHolding = 60;
            int chileHolding = 40;

            try {
                // Create Factory for WMQ JMS provider
                JmsFactoryFactory ff = JmsFactoryFactory.getInstance(WMQConstants.WMQ_PROVIDER);

                // Create connection factory
                cf = ff.createConnectionFactory();
                // Set MQ properties
                cf.setStringProperty(WMQConstants.WMQ_QUEUE_MANAGER, "QM3");
                cf.setIntProperty(WMQConstants.WMQ_CONNECTION_MODE, WMQConstants.WMQ_CM_BINDINGS);

                // Create message context
                msgContext = cf.createContext();

                // Create a topic destination
                Topic fifaScores = msgContext.createTopic("/FIFA2014/UPDATES");

                // Create publisher to publish updates from stadium
                JMSProducer msgProducer = msgContext.createProducer();

                while(true){
                    // Send match updates
                    switch(switchIndex){
                        // Attendance
                    case 0:
                        msgBody ="Stadium Attendence " + stadiumAttendence;
                        stadiumAttendence += 314;
                        break;

                        // Goals
                    case 1:
                        msgBody ="SCORE: The Netherlands: " + nederlandsGoals + " - Chile:" + chileGoals;
                        break;
```

```
                        // Ball possession percentage
                    case 2:
                        msgBody ="Ball possession: The Netherlands: " + nederlandsHolding + "% - Chile:
" + chileHolding + "%";
                        if((nederlandsHolding > 60) && (nederlandsHolding < 70)){
                            nederlandsHolding -= 2;
                            chileHolding += 2;
                        }else{
                            nederlandsHolding += 2;
                            chileHolding -= 2;
                        }
                        break;
                    }

                    // Publish and wait for two seconds to publish next update
                    msgProducer.send (fifaScores, msgBody);
                    try{
                        Thread.sleep(2000);
                    }catch(InterruptedException iex){

                    }

                    // Increment and reset the index if greater than 2
                    switchIndex++;
                    if(switchIndex > 2)
                        switchIndex = 0;
                }
            }catch(JMSException jmsEx){
                System.out.println(jmsEx);
            }
        }

    /*
     * (non-Javadoc)
     * @see java.lang.Runnable#run()
     */
    public void run() {
        // If this is a publisher thread
        if(threadName == "PUBLISHER"){
            matchUpdatePublisher();
        }else{
            // Create subscription and start receiving publications
            sharedNonDurableSubscriptionDemo();
        }
    }

       // Start thread
   public void start (){
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```
/*
* Demonstrate JMS 2.0 and later simplified API using IBM MQ 91 JMS Implementation
*/
public class Mqv91jms2Sample {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // Create first subscriber and start
        SharedNonDurableSubscriberAndPublisher subOne = new
SharedNonDurableSubscriberAndPublisher( "SUB1");
        subOne.start();

        // Create second subscriber and start
        SharedNonDurableSubscriberAndPublisher subTwo = new
SharedNonDurableSubscriberAndPublisher( "SUB2");
        subTwo.start();

        // Create third subscriber and start
        SharedNonDurableSubscriberAndPublisher subThree = new
SharedNonDurableSubscriberAndPublisher( "SUB3");
        subThree.start();
```

```
        // Create publisher and start
        SharedNonDurableSubscriberAndPublisher publisher = new
SharedNonDurableSubscriberAndPublisher( "PUBLISHER");
        publisher.start();
    }
}
```

Related concepts

IBM MQ Java language interfaces

V 9.4.0 Configuring your modular application to use IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging

V 9.4.0 You can use IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging in a modular manner by requiring the appropriate module within your application, and including the appropriate directory in the module-path.

The modular packaging

The unified JAR files for IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging provide automatic module names, which replace the default names that are derived from the JAR file names.

- IBM MQ classes for JMS (`com.ibm.mq.allclient.jar`) are provided with a module name of `com.ibm.mq.javax`.
- IBM MQ classes for Jakarta Messaging (`com.ibm.mq.jakarta.client.jar`) are provided with a module name of `com.ibm.mq.jakarta`.

The default `MQ_HOME/java/lib` directory is unsuitable for modular use because modules cannot contain the same package, and the default directory contains the same packages in multiple JARs. Therefore, new directories are available for that contain only the JAR files that are needed, with no duplication of packages between the JARs. These directories are suitable for inclusion on a `module-path`.

Note: If you have applications that use the available JAR files in a modular context by relying on the default module names, you must update your applications to require the new module names. The default module names are derived from the JAR file names.

Configuring your modular application to use IBM MQ classes for JMS

You can configure your modular application to use IBM MQ classes for JMS (`com.ibm.mq.allclient.jar`) by completing the following steps:

- Configure the application to require the `com.ibm.mq.javax` module.
- Configure the application to include the `MQ_HOME/java/lib/modules/javax` directory in the module-path.

Configuring your modular application to use IBM MQ classes for Jakarta Messaging

You can configure your modular application to use IBM MQ classes for Jakarta Messaging (`com.ibm.mq.jakarta.client.jar`) by completing the following steps:

- Configure the application to require the `com.ibm.mq.jakarta` module.
- Configure the application to include the `MQ_HOME/java/lib/modules/jakarta` directory in the module-path.

Configuring your modular application to use IBM MQ classes for Java

To use IBM MQ classes for Java from a modular application, you can use either the configuration for IBM MQ classes for JMS or the configuration for IBM MQ classes for Jakarta Messaging, as both client JAR files

support IBM MQ classes for Java. However, your application must use only one of these configurations, not both.

IBM MQ classes for JMS Application Server Facilities

This topic describes how IBM MQ classes for JMS implements the ConnectionConsumer class and advanced functionality in the Session class. It also summarizes the function of a server session pool.

Important: This information is for reference only. An application must not be written to use this interface: it is used within the IBM MQ resource adapter to connect to Java EE servers. For practical connection information, see "Using the IBM MQ resource adapter" on page 420.

IBM MQ classes for JMS supports the Application Server Facilities (ASF) that are specified in the *Java Message Service Specification* (see Oracle Technology Network for Java Developers). This specification identifies three roles within this programming model:

- **The JMS provider** supplies ConnectionConsumer and advanced Session functionality.
- **The application server** supplies ServerSessionPool and ServerSession functionality.
- **The client application** uses the functionality that the JMS provider and application server supply.

The information in this topic does not apply if an application uses a real-time connection to a broker.

The JMS ConnectionConsumer

The ConnectionConsumer interface provides a high-performance method to deliver messages concurrently to a pool of threads.

The JMS specification enables an application server to integrate closely with a JMS implementation by using the `ConnectionConsumer` interface. This feature provides concurrent processing of messages. Typically, an application server creates a pool of threads, and the JMS implementation makes messages available to these threads. A JMS-aware application server (such as WebSphere Application Server) can use this feature to provide high-level messaging functionality, such as message driven beans.

Normal applications do not use the ConnectionConsumer, but expert JMS clients might use it. For such clients, the ConnectionConsumer provides a high-performance method to deliver messages concurrently to a pool of threads. When a message arrives on a queue or a topic, JMS selects a thread from the pool and delivers a batch of messages to it. To do this, JMS runs an associated MessageListener's `onMessage()` method.

You can achieve the same effect by constructing multiple Session and MessageConsumer objects, each with a registered MessageListener. However, the ConnectionConsumer provides better performance, less use of resources, and greater flexibility. In particular, fewer Session objects are required.

Planning an application with ASF

This section tells you how to plan an application including:

- "General principles for point-to-point messaging using ASF" on page 322
- "General principles for publish/subscribe messaging using ASF" on page 323
- "Removing messages from the queue in ASF" on page 324
- Handling poison messages in ASF. See "Handling poison messages in IBM MQ classes for JMS" on page 225.

General principles for point-to-point messaging using ASF

Use this topic for general information about point-to-point messaging using ASF.

When an application creates a ConnectionConsumer from a QueueConnection object, it specifies a JMS queue object and a selector string. The ConnectionConsumer then begins to provide messages to sessions in the associated ServerSessionPool. Messages arrive on the queue, and if they match the selector, they are delivered to sessions in the associated ServerSessionPool.

In IBM MQ terms, the queue object refers to either a QLOCAL or a QALIAS on the local queue manager. If it is a QALIAS, that QALIAS must refer to a QLOCAL. The fully resolved IBM MQ QLOCAL is known

as the *underlying QLOCAL*. A ConnectionConsumer is said to be *active* if it is not closed and its parent QueueConnection is started.

It is possible for multiple ConnectionConsumers, each with different selectors, to run against the same underlying QLOCAL. To maintain performance, unwanted messages must not accumulate on the queue. Unwanted messages are those for which no active ConnectionConsumer has a matching selector. You can set the QueueConnectionFactory so that these unwanted messages are removed from the queue (for details, see "Removing messages from the queue in ASF" on page 324). You can set this behavior in one of two ways:

- Use the JMS administration tool to set the QueueConnectionFactory to MRET(NO).
- In your program, use:

```
MQQueueConnectionFactory.setMessageRetention(WMQConstants.WMQ_MRET_NO)
```

If you do not change this setting, the default is to retain such unwanted messages on the queue.

When you set up the IBM MQ queue manager, consider the following points:

- The underlying QLOCAL must be enabled for shared input. To do this, use the following MQSC command:

```
ALTER QLOCAL( your.qlocal.name ) SHARE GET(ENABLED)
```

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:

```
ALTER QMGR DEADQ( your.dead.letter.queue.name )
```

- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the IBM MQ documentation for your specific platform.
- If unwanted messages are left on the queue, they degrade the system performance. Therefore, plan your message selectors so that between them, the ConnectionConsumers will remove all messages from the queue.

For details about MQSC commands, see MQSC commands.

General principles for publish/subscribe messaging using ASF
ConnectionConsumers receive messages for a specified Topic. A ConnectionConsumer can be durable or non-durable. You must specify which queue or queues the ConnectionConsumer uses.

When an application creates a ConnectionConsumer from a TopicConnection object, it specifies a Topic object and a selector string. The ConnectionConsumer then begins to receive messages that match the selector on that Topic , including any retained publications for the topic subscribed to.

Alternatively, an application can create a durable ConnectionConsumer that is associated with a specific name. This ConnectionConsumer receives messages that have been published on the Topic since the durable ConnectionConsumer was last active. It receives all such messages that match the selector on the Topic. However, if the ConnectionConsumer is using read-ahead, it can lose nonpersistent messages that are in the client buffer when it closes.

If IBM MQ classes for JMS is in IBM MQ messaging provider migration mode, a separate queue is used for non-durable ConnectionConsumer subscriptions. The CCSUB configurable option on the TopicConnectionFactory specifies the queue to use. Normally, the CCSUB specifies a single queue for use by all ConnectionConsumers that use the same TopicConnectionFactory. However, it is possible to make each ConnectionConsumer generate a temporary queue by specifying a queue name prefix followed by an asterisk (*).

If IBM MQ classes for JMS is in IBM MQ messaging provider migration mode, the CCDSUB property of the Topic specifies the queue to use for durable subscriptions. Again, this can be a queue that already exists or a queue name prefix followed by an asterisk (*). If you specify a queue that already exists, all durable ConnectionConsumers that subscribe to the Topic use this queue. If you specify a queue name prefix followed by an asterisk (*), a queue is generated the first time that a durable ConnectionConsumer is created with a particular name. This queue is reused later when a durable ConnectionConsumer is created with the same name.

When you set up the IBM MQ queue manager, consider the following points:

- Your queue manager must have an enabled dead-letter queue. If a ConnectionConsumer experiences a problem when it puts a message on the dead-letter queue, message delivery from the underlying QLOCAL stops. To define a dead-letter queue, use:

```
ALTER QMGR DEADQ( your.dead.letter.queue.name )
```

- The user that runs the ConnectionConsumer must have authority to perform MQOPEN with MQOO_SAVE_ALL_CONTEXT and MQOO_PASS_ALL_CONTEXT. For details, see the IBM MQ documentation for your platform.
- You can optimize performance for an individual ConnectionConsumer by creating a separate, dedicated, queue for it. This is at the cost of extra resource usage.

Removing messages from the queue in ASF
When an application uses ConnectionConsumers, JMS might need to remove messages from the queue in a number of situations.

These situations are as follows:

Badly formatted message
A message might arrive that JMS cannot parse.

Poison message
A message might reach the backout threshold, but the ConnectionConsumer fails to requeue it on the backout queue.

No interested ConnectionConsumer
For point-to-point messaging, when the QueueConnectionFactory is set so that it does not retain unwanted messages, a message arrives that is unwanted by any of the ConnectionConsumers.

In these situations, the ConnectionConsumer attempts to remove the message from the queue. The disposition options in the report field of the message's MQMD set the exact behavior. These options are:

MQRO_DEAD_LETTER_Q
The message is requeued to the queue manager's dead-letter queue. This is the default.

MQRO_DISCARD_MSG
The message is discarded.

The ConnectionConsumer also generates a report message, and this also depends on the report field of the message's MQMD. This message is sent to the message's ReplyToQ on the ReplyToQmgr. If there is an error while the report message is being sent, the message is sent to the dead-letter queue instead. The exception report options in the report field of the message's MQMD set details of the report message. These options are:

MQRO_EXCEPTION
A report message is generated that contains the MQMD of the original message. It does not contain any message body data.

MQRO_EXCEPTION_WITH_DATA
A report message is generated that contains the MQMD, any MQ headers, and 100 bytes of body data.

MQRO_EXCEPTION_WITH_FULL_DATA
A report message is generated that contains all data from the original message.

default
No report message is generated.

When report messages are generated, the following options are honored:

- MQRO_NEW_MSG_ID
- MQRO_PASS_MSG_ID
- MQRO_COPY_MSG_ID_TO_CORREL_ID
- MQRO_PASS_CORREL_ID

If a poison message cannot be requeued, perhaps because the dead-letter queue is full or authorization is wrongly specified, what happens depends on the persistence of the message. If the message is nonpersistent, the message is discarded and no report message is generated. If the message is persistent, delivery of messages to all connection consumers listening on that destination stops. Such connection consumers must be closed and the problem resolved before they can be re-created and message delivery restarted.

It is important to define a dead-letter queue, and to check it regularly to ensure that no problems occur. Particularly, ensure that the dead-letter queue does not reach its maximum depth, and that its maximum message size is large enough for all messages.

When a message is requeued to the dead-letter queue, it is preceded by an IBM MQ dead-letter header (MQDLH). See MQDLH - Dead-letter header for details about the format of the MQDLH. You can identify messages that a ConnectionConsumer has placed on the dead-letter queue, or report messages that a ConnectionConsumer has generated, by the following fields:

- PutApplType is `MQAT_JAVA (0x1C)`
- PutApplName is "`MQ JMS ConnectionConsumer`"

These fields are in the MQDLH of messages on the dead-letter queue, and the MQMD of report messages. The feedback field of the MQMD, and the Reason field of the MQDLH, contain a code describing the error. For details about these codes, see "Reason and feedback codes in ASF" on page 326. Other fields are as described in MQDLH - Dead-letter header.

Handling poison messages in ASF
Within the Application Server Facilities, poison message handling is handled slightly differently to elsewhere in IBM MQ classes for JMS.

For information about poison message handling in IBM MQ classes for JMS, see "Handling poison messages in IBM MQ classes for JMS" on page 225.

When you use Application Server Facilities (ASF), the ConnectionConsumer, rather than the MessageConsumer, processes poison messages. The ConnectionConsumer requeues messages according to the BackoutThreshold and BackoutRequeueQName properties of the queue.

When an application uses ConnectionConsumers, the circumstances in which a message is backed out depend on the session that the application server provides:

- When the session is non-transacted, with AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE, a message is backed out only after a system error, or if the application terminates unexpectedly.
- When the session is non-transacted with CLIENT_ACKNOWLEDGE, unacknowledged messages can be backed out by the application server calling Session.recover().

  Typically, the client implementation of MessageListener or the application server calls Message.acknowledge(). Message.acknowledge() acknowledges all messages delivered on the session so far.
- When the session is transacted, unacknowledged messages can be backed out by the application server calling Session.rollback().
- If the application server supplies an XASession, messages are committed or backed out depending on a distributed transaction. The application server takes responsibility for completing the transaction.

Related concepts

"Handling poison messages in IBM MQ classes for JMS" on page 225
A poison message is one which cannot be processed by a receiving application. If a poison message is delivered to an application and rolled back a specified number of times, the IBM MQ classes for JMS can move it to a backout queue.

Error handling

This section covers various aspects of error handling, including "Recovering from error conditions in the ASF" on page 326 and "Reason and feedback codes in ASF" on page 326.

Recovering from error conditions in the ASF

If a ConnectionConsumer experiences a serious error, message delivery to all ConnectionConsumers with an interest in the same QLOCAL stops. When this occurs, any ExceptionListener that is registered with the affected Connection is notified. There are two ways in which an application can recover from these error conditions.

Typically, a serious error of this nature occurs if the ConnectionConsumer cannot requeue a message to the dead-letter queue, or it experiences an error when reading messages from the QLOCAL.

Because any ExceptionListener that is registered with the affected Connection is notified, you can use them to identify the cause of the problem. In some cases, the system administrator must intervene to resolve the problem.

Use one of the following techniques to recover from these error conditions:

- Call `close()` on all affected ConnectionConsumers. The application can create new ConnectionConsumers only after all affected ConnectionConsumers are closed and any system problems are resolved.
- Call `stop()` on all affected Connections. After all Connections are stopped and any system problems are resolved, the application can `start()` its Connections successfully.

Reason and feedback codes in ASF

Use reason and feedback codes to determine the cause of an error. Common reason codes generated by the ConnectionConsumer are given here.

To determine the cause of an error, use the following information:

- The feedback code in any report messages
- The reason code in the MQDLH of any messages in the dead-letter queue

ConnectionConsumers generate the following reason codes.

MQRC_BACKOUT_THRESHOLD_REACHED (0x93A; 2362)

Cause

The message has reached the Backout Threshold defined on the QLOCAL, but no Backout Queue is defined.

On platforms where you cannot define the Backout Queue, the message has reached the JMS-defined backout threshold of 20.

Action

If this is not wanted, define the Backout Queue for the relevant QLOCAL. Also look for the cause of the multiple backouts.

MQRC_MSG_NOT_MATCHED (0x93B; 2363)

Cause

In point-to-point messaging, there is a message that does not match any of the selectors for the ConnectionConsumers monitoring the queue. To maintain performance, the message is requeued to the dead-letter queue.

Action

To avoid this situation, ensure that ConnectionConsumers using the queue provide a set of selectors that deal with all messages, or set the QueueConnectionFactory to retain messages.

Alternatively, investigate the source of the message.

MQRC_JMS_FORMAT_ERROR (0x93C; 2364)

Cause

JMS cannot interpret the message on the queue.

Action

Investigate the origin of the message. JMS normally delivers messages of an unexpected format as a BytesMessage or TextMessage. Occasionally, this fails if the message is very badly formatted.

Other codes that appear in these fields are caused by a failed attempt to requeue the message to a Backout Queue. In this situation, the code describes the reason that the requeue failed. To diagnose the cause of these errors, refer to API>API completion and reason codes.

If the report message cannot be put on the ReplyToQ, it is put on the dead-letter queue. In this situation, the feedback field of the MQMD is completed as described in this topic. The reason field in the MQDLH explains why the report message could not be placed on the ReplyToQ.

The function of a server session pool in AFS

This topic summarizes the function of a server session pool.

Figure 45 on page 328 summarizes the principles of ServerSessionPool and ServerSession functionality.

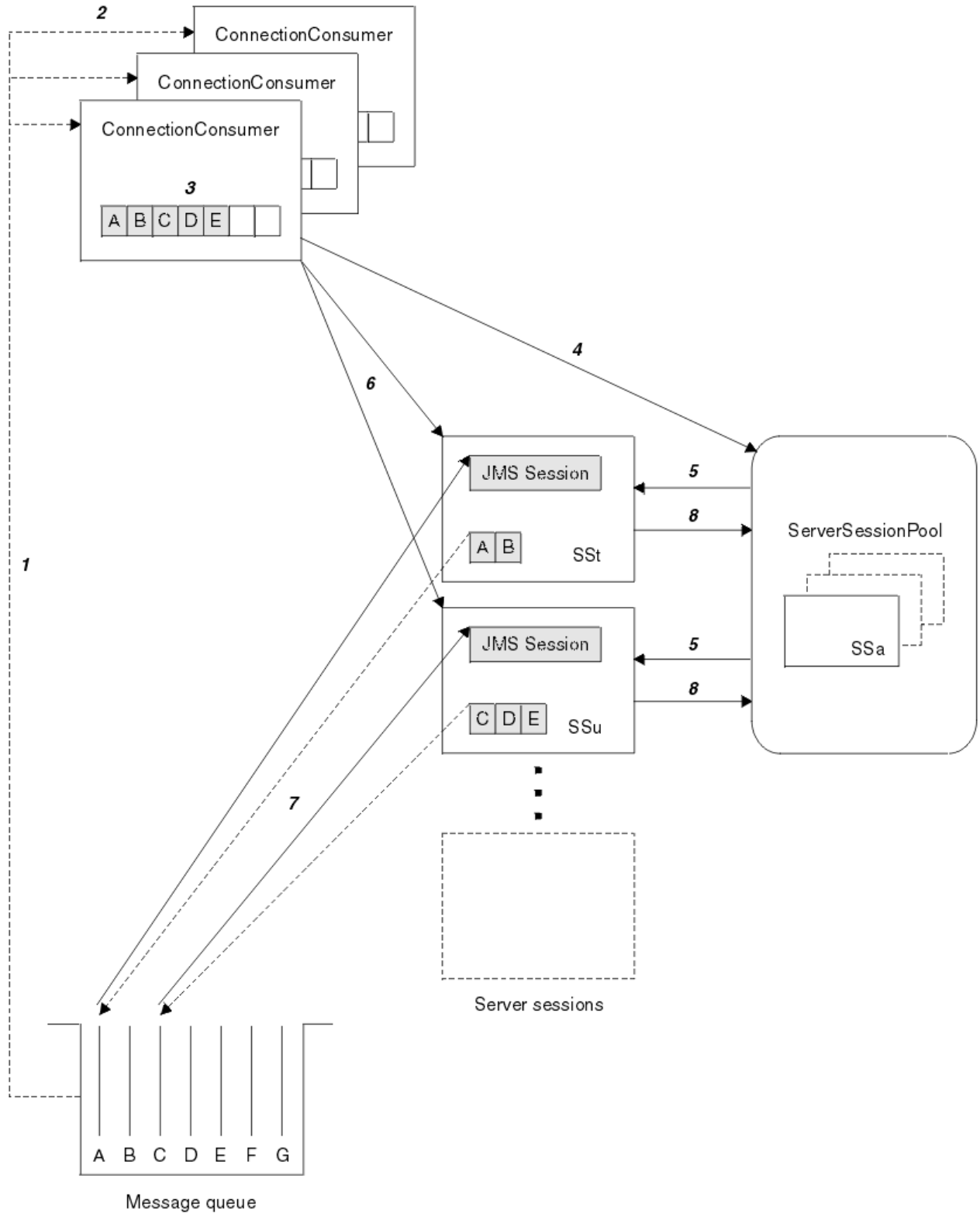


Figure 45. ServerSessionPool and ServerSession functionality

1. The ConnectionConsumers get message references from the queue.
2. Each ConnectionConsumer selects specific message references.
3. The ConnectionConsumer buffer holds the selected message references.
4. The ConnectionConsumer requests one or more ServerSessions from the ServerSessionPool.

5. ServerSessions are allocated from the ServerSessionPool.
6. The ConnectionConsumer assigns message references to the ServerSessions and starts the ServerSession threads running.
7. Each ServerSession retrieves its referenced messages from the queue. It passes them to the onMessage method from the MessageListener that is associated with the JMS Session.
8. After it completes its processing, the ServerSession is returned to the pool.

An application server normally supplies ServerSessionPool and ServerSession functionality.

z/OS Using IBM MQ classes for JMS in a CICS Liberty JVM server

Java programs running in a CICS Liberty JVM server can use the IBM MQ classes for JMS to access IBM MQ.

You must be using an IBM MQ 9.1.0 or later version of the IBM MQ resource adapter. You can obtain the resource adapter from Fix Central (see "Installing the resource adapter in Liberty" on page 429).

There are two flavors of Liberty Profile JVMs available in CICS 5.3 and later, the types of connection possible to IBM MQ are restricted as follows:

CICS Liberty Standard

- The IBM MQ resource adapter can connect to any in-service version of IBM MQ in CLIENT mode
- The IBM MQ resource adapter can connect to any in-service version of IBM MQ for z/OS in BINDINGS mode when there is no CICS connection (active CICS MQCONN resource definition) to the same queue manager from the same CICS region.

CICS Liberty Integrated

- The IBM MQ resource adapter can connect to any in-service version of IBM MQ in CLIENT mode.
- BINDINGS mode connection is not supported.

For details on setting up and configuring your system, see Using IBM MQ classes for JMS in a Liberty JVM server in the CICS documentation.

Using IBM MQ classes for JMS/ Jakarta Messaging in IMS

Standards-based messaging support within an IMS environment is provided through the use of IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging.

Check the system requirements for the IMS system that your enterprise uses. See IMS 15.2 for further information.

This set of topics describes how to set up the IBM MQ classes for JMS in an IMS environment, and the API restrictions that apply when using the classic (JMS 1.1) and simplified (JMS 2.0) interfaces. See "JMS API restrictions" on page 334 for a list of the API-specific information.

Note: Similar restrictions apply to the legacy (JMS 1.0.2) domain-specific interfaces, but they are not specifically described here.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. It is not supported to use both the Jakarta Messaging 3.0 API and the JMS 2.0 API in the same application. For more information, see Using IBM MQ classes for JMS/Jakarta Messaging.

Supported IMS dependent regions

The following dependent region types are supported:

- MPR
- BMP
- IFP

- JMP 31 and 64 bit Java virtual machines (JVMs)
- JBP 31 and 64 bit JVMs

Unless specifically mentioned in the following topics, IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging behave the same in all region types.

Supported Java Virtual Machines

IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging require IBM Runtime Environment, Java Technology Edition 8. IBM Semeru Runtime Certified Edition for z/OS, Version 11 is not supported.

Other restrictions

The following restrictions apply when using IBM MQ classes for JMS in an IMS environment:

- Client mode connections are not supported.
- Connections are only supported to IBM MQ 8.0 queue managers using the IBM MQ messaging provider `Normal`, mode.

  The **PROVIDERVERSION** attribute on the connection factory must be either unspecified, or a value greater than, or equal to, seven.
- Use of any of the XA connection factories, for example `com.ibm.mq.jms.MQXAConnectionFactory`, is not supported.

Related tasks

Defining IBM MQ to IMS

Setting up the IMS adapter for use with IBM MQ classes for JMS/Jakarta Messaging

IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging make use of the same IBM MQ-IMS adapter as used by other programming languages. This adapter uses the IMS External Subsystem Attach Facility (ESAF).

Before you begin

Before completing the following procedure, you must configure the IMS adapter for the relevant queue managers, and IMS control and dependent regions, as described in Setting up the IMS adapter.

Attention: You do not need to perform the step that describes building a dynamic stub, unless you need the dynamic stub for other purposes.

After you have configured the IMS adapter, carry out the following procedure.

Procedure

1. Update the LIBPATH variable in the member of your IMS PROCLIB that is referenced by the ENVIRON parameter in your dependent region JCL (for example, DFSJVMEV) so that it includes the IBM MQ classes for JMS native libraries.

   That is, the zFS directory that contains `libmqjims.so`. For example, DFSJVMEV might look like the following, where the last line is the directory containing the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging native libraries:

```
LIBPATH=>
/java/latest/bin/j9vm:>
/java/latest/bin:>
/ims/latest/dbdc/imsjava/classic/lib:>
/ims/latest/dbdc/imsjava/lib:>
/mqm/latest/java/lib
```

2. Add the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging to the class path of the JVM, used by your IMS dependent region, by updating the `java.class.path` option.

Do this by following the instructions in DFSJVMMS member of the IMS PROCLIB data set.

For example, you can use the following, where the line in bold indicates the update:

JMS 3.0

```
-Djava.class.path=/ims/latest/dbdc/imsjava/imsutm.jar:/ims/latest/dbdc/imsjava/imsudb.jar:
/mqm/latest/java/lib/com.ibm.mq.jakarta.client.jar
```

JMS 2.0

```
-Djava.class.path=/ims/latest/dbdc/imsjava/imsutm.jar:/ims/latest/dbdc/imsjava/imsudb.jar:
/mqm/latest/java/lib/com.ibm.mq.allclient.jar
```

Note: While there are many different jar files available in the directory containing the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging, you need only `com.ibm.mq.allclient.jar` (JMS 2.0) or `com.ibm.mq.jakarta.client.jar` (Jakarta Messaging 3.0).

3. Stop and restart any IMS dependent regions that will make use of the IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging.

What to do next

Create and configure connection factories and destinations.

There are three possible approaches for instantiating the IBM MQ implementations of connection factories and destinations. See “Creating and configuring connection factories and destinations” on page 197 for details.

Note that these three approaches are all valid in an IMS environment.

Related concepts

Setting up the IMS adapter

Defining IBM MQ to IMS

Transactional behavior

Messages sent and received by the IBM MQ classes for JMS in an IMS environment are always associated with the IMS unit of work (UOW) that is active on the current task.

That UOW can only be completed by calling the commit or rollback methods on an instance of the `com.ibm.ims.dli.tm.Transaction` object, or by the IMS task ending normally in which case the UOW is implicitly committed. If the IMS task ends abnormally, the UOW is rolled back.

As a result of this, the values of the **`transacted`** and **`acknowledgeMode`** arguments are ignored when calling any of the `Connection.createSession`, or `ConnectionFactory.createContext` methods. Additionally the following methods are not supported. Calling any of the following methods results in an `IllegalStateException` in the session case:

- `javax.jms.Session.commit()`
- `javax.jms.Session.recover()`
- `javax.jms.Session.rollback()`

and an `IllegalStateRuntimeSession` in the JMS context case:

- `javax.jms.JMSContext.commit()`
- `javax.jms.JMSContext.recover()`
- `javax.jms.JMSContext.rollback()`

There is one exception to this behavior. If a session or JMS context is created using one of the following mechanisms:

- `Connection.createSession(false, Session.AUTO_ACKNOWLEDGE)`
- `Connection.createSession(Session.AUTO_ACKNOWLEDGE)`
- `ConnectionFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)`

then the behavior of that session, or JMS context, is as follows:

- Any messages sent, are sent outside of the IMS UOW. That is, they will be available on the target destination immediately, or when the provided delivery delay interval has completed.
- Any non-persistent messages will be received outside of the IMS UOW, unless the SYNCPOINTALLGETS property has been specified on the connection factory that created the session or JMS context.
- Persistent messages will always be received inside the IMS UOW.

This might be useful if, for example, you want to write an audit message to a queue even if the UOW rolls back.

Implications of IMS syncpoints

The IBM MQ classes for JMS build upon the existing IBM MQ adapter support which makes use of ESAF. This means that the documented behavior applies, including all open handles being closed by the IMS adapter when a syncpoint occurs.

z/OS See "Syncpoints in IMS applications" on page 67 for more information.

To illustrate this point, consider the following code running in a JMP environment. The second call to `mp.send()` results in a `JMSException` as the `messageQueue.getUnique(inputMessage)` code results in all open IBM MQ connection and object handles being closed.

Similar behavior is observed if the `getUnique()` call was replaced with `Transaction.commit()`, but not if `Transaction.rollback()` was used.

```
//Create a connection to queue manager MQ21.
MQConnectionFactory cf = new MQConnectionFactory();
cf.setQueueManager("MQ21");

Connection c = cf.createConnection();
Session s = c.createSession();

//Send a message to MQ queue Q1.
Queue q = new MQQueue("Q1");
MessageProducer mp = s.createProducer(q);
TextMessage m = s.createTextMessage("Hello world!");
mp.send(m);

//Get a message from an IMS message queue. This results in a GU call
//which results in all MQ handles being closed.
Application a = ApplicationFactory.createApplication();
MessageQueue messageQueue = a.getMessageQueue();
IOMessage inputMessage = a.getIOMessage(MESSAGE_CLASS_NAME);
messageQueue.getUnique(inputMessage);

//This attempt to send another message will result in a JMSException containing a
//MQRC_HCONN_ERROR as the connection/handle has been closed.
mp.send(m);
```

The correct code to use in this scenario is as follows. In this case the connection to IBM MQ is closed prior to calling `getUnique()`. The connection and session are then re-created in order to send another message.

```
//Create a connection to queue manager MQ21.
MQConnectionFactory cf = new MQConnectionFactory();
cf.setQueueManager("MQ21");

Connection c = cf.createConnection();
Session s = c.createSession();

//Send a message to MQ queue Q1.
Queue q = new MQQueue("Q1");
MessageProducer mp = s.createProducer(q);
TextMessage m = s.createTextMessage("Hello world!");
mp.send(m);

//Close the connection to MQ, which closes all MQ object handles.
//The send of the message will be committed by the subsequent GU call.
c.close();
```

```
c = null;
s = null;
mp = null;

//Get a message from an IMS message queue. This results in a GU call.
Application a = ApplicationFactory.createApplication();
MessageQueue messageQueue = a.getMessageQueue();
IOMessage inputMessage = a.getIOMessage(MESSAGE_CLASS_NAME);
messageQueue.getUnique(inputMessage);

//Re-create the connection to MQ and send another message;
c = cf.createConnection();
s = c.createSession();
mp = s.createProducer(q);
m = s.createTextMessage("Hello world 2!");
mp.send(m);
```

Considerations when using the IMS adapter

You need to be aware of the following restrictions. You can have only one connection handle for each queue manager. There are implications in the interaction with IBM MQ when using both JMS and native code. There are limitations to connection authentication and authorization.

One connection handle for each queue manager

Only one connection handle at a time to a specific queue manager is allowed in IMS dependent regions. Any subsequent attempts to connect to the same queue manager reuse the existing handle.

While this behavior should not cause any problems in an application that only uses the IBM MQ classes for JMS, this behavior can cause problems in applications that interact with IBM MQ, when using both the IBM MQ classes for JMS and the MQI in native code written in languages, such as COBOL or C.

Implications of interacting with IBM MQ when using both JMS and native code

Problems can occur when interleaving Java code and native code that both use IBM MQ functionality and when the connection to IBM MQ is not closed before leaving either the native or Java code.

For example, in the following pseudo code, a connection handle to a queue manager is originally established in Java code using the IBM MQ classes for JMS. The connection handle is reused in COBOL code and invalidated by a call to MQDISC.

The next time the IBM MQ classes for JMS make use of the connection handle a `JMSException` with a reason code of MQRC_HCONN_ERROR results.

```
COBOL code running in message processing region
    Use the Java Native Interface (JNI) to call Java code
       Create MQ connection and session - this creates an MQ connection handle
       Send message to MQ queue
       Store connection and session in static variable
       Return to COBOL code

    MQCONN - picks up MQ connection handle established in Java code
    MQDISC - invalidates connection handle

    Use the Java Native Interface (JNI) to call Java code
       Get session from static variable
       Create a message consumer - fails as connection handle invalidated
```

There are other similar usage patterns which can result in MQRC_HCONN_ERROR.

While it is possible to share IBM MQ connection handles between native and Java code (for example, the previous example would work if there had not been an MQDISC call) in general, the best practice is to close any connection handles before changing from Java to native code, or the other way round.

Connection authentication and authorization

The JMS specification allows a user name and password to be specified for authentication and authorization when creating a connection or JMS context object.

This is not supported in an IMS environment. Attempting to create a connection while specifying a user name and password results in a JMS Exception being thrown. Attempting to create a JMS context, while specifying a user name and password, results in a JMSRuntimeException being thrown.

Instead, existing mechanisms for authentication and authorization when connecting to IBM MQ from an IMS environment must be used.

For more information, see Setting up security on z/OS. In particular, refer to User IDs for security checking, which describes the user IDs that can be used.

Related tasks

Setting up security on z/OS

JMS API restrictions

From a JMS specification perspective, the IBM MQ classes for JMS treat IMS as a Java EE or Jakarta EE compliant application server, that always has a JTA transaction in progress.

For example, you can never call `javax.jms.Session.commit()` in IMS, because the JMS specification states that you can not call it in a JEE EJB, or Web container, while a JTA transaction is in progress.

This results in the following restrictions to the JMS API, in addition to those described in "Transactional behavior" on page 331.

Classic API restrictions

- `javax.jms.Connection.createConnectionConsumer(javax.jms.Destination, String, javax.jms.ServerSessionPool, int)` always throws a `JMSException`.
- `javax.jms.Connection.createDurableConnectionConsumer(javax.jms.Topic, String, String, javax.jms.ServerSessionPool, int)` always throws a `JMSException`.
- All three variants of `javax.jms.Connection.createSession` always throws a `JMSException` if the connection already has an existing session active.
- `javax.jms.Connection.createSharedConnectionConsumer(javax.jms.Topic, String, String, javax.jms.ServerSessionPool, int)` always throws a `JMSException`.
- `javax.jms.Connection.createSharedDurableConnectionConsumer(javax.jms.Topic, String, String, javax.jms.ServerSessionPool, int)` always throws a `JMSException`.
- `javax.jms.Connection.setClientID()` always throws a `JMSException`.
- `javax.jms.Connection.setExceptionListener(javax.jms.ExceptionListener)` always throws a `JMSException`.
- `javax.jms.Connection.stop()` always throws a `JMSException`.
- `javax.jms.MessageConsumer.setMessageListener(javax.jms.MessageListener)` always throws a `JMSException`.
- `javax.jms.MessageConsumer.getMessageListener()` always throws a `JMSException`.
- `javax.jms.MessageProducer.send(javax.jms.Destination, javax.jms.Message,javax.jms.CompletionListener)` always throws a `JMSException`.
- `javax.jms.MessageProducer.send(javax.jms.Destination, javax.jms.Message, int, int, long, javax.jms.CompletionListener)` always throws a `JMSException`.
- `javax.jms.MessageProducer.send(javax.jms.Message, int, int, long, javax.jms.CompletionListener)` always throws a `JMSException`.
- `javax.jms.MessageProducer.send(javax.jms.Message,javax.jms.CompletionListener)` always throws a `JMSException`.
- `javax.jms.Session.run()` always throws a `JMSRuntimeException`.
- `javax.jms.Session.setMessageListener(javax.jms.MessageListener)` always throws a `JMSException`.
- `javax.jms.Session.getMessageListener()` always throws a `JMSException`.

Simplified API restrictions

- `javax.jms.JMSContext.createContext(int)` always throws a `JMSRuntimeException`.
- `javax.jms.JMSContext.setClientID(String)` always throws a `JMSRuntimeException`.
- `javax.jms.JMSContext.setExceptionListener(javax.jms.ExceptionListener)` always throws a `JMSRuntimeException`.
- `javax.jms.JMSContext.stop()` always throws a `JMSRuntimeException`.
- `javax.jms.JMSProducer.setAsync(javax.jms.CompletionListener)` always throws a `JMSRuntimeException`.

Using IBM MQ classes for Java

Use IBM MQ in a Java environment. IBM MQ classes for Java allow a Java application to connect to IBM MQ as an IBM MQ client, or connect directly to an IBM MQ queue manager.

Note:

Stabilized IBM will make no further enhancements to the IBM MQ classes for Java and they are functionally stabilized at the level shipped in IBM MQ 8.0. Existing applications that use the IBM MQ classes for Java continue to be fully supported, but new features will not be added and requests for enhancements will be rejected. Fully supported means that defects will be fixed together with any changes necessitated by changes to IBM MQ System Requirements.

The IBM MQ classes for Java are not supported in IMS.

The IBM MQ classes for Java are not supported in WebSphere Liberty. They must not be used with either the IBM MQ Liberty messaging feature, or with the generic JCA support. For more information, see Using WebSphere MQ Java Interfaces in J2EE/JEE Environments.

IBM MQ classes for Java is one of three alternative APIs that Java applications can for use to access IBM MQ resources. The other APIs are:

- JMS 3.0 IBM MQ classes for Jakarta Messaging
- JMS 2.0 IBM MQ classes for JMS

For more information, see “Accessing IBM MQ from Java - Choice of API” on page 82.

From IBM MQ 9.3, the IBM MQ classes for Java are built with Java 8. The Java 8 runtime environment supports running earlier class file versions.

IBM MQ classes for Java encapsulate the Message Queue Interface (MQI), the native IBM MQ API and use a similar object model to the C++ and .NET interfaces to IBM MQ.

Programmable options allow IBM MQ classes for Java to connect to IBM MQ in either of the following ways:

- In client mode as an IBM MQ MQI client by using Transmission Control Protocol/Internet Protocol (TCP/IP)
- In bindings mode, connecting directly to IBM MQ by using the Java Native Interface (JNI)

Note: Automatic client reconnection is not supported by IBM MQ classes for Java.

Client mode connection

An IBM MQ classes for Java application can connect to any supported queue manager by using client mode.

To connect to a queue manager in client mode, an IBM MQ classes for Java application can run on the same system on which the queue manager is running, or on a different system. In each case, IBM MQ classes for Java connects to the queue manager over TCP/IP.

For more information on how to write applications to use client mode connections, see “IBM MQ classes for Java connection modes” on page 359.

Bindings mode connection

When used in bindings mode, IBM MQ classes for Java uses the Java Native Interface (JNI) to call directly into the existing queue manager API, rather than communicating through a network. In most environments, connecting in bindings mode provides better performance for IBM MQ classes for Java applications than connecting in client mode, by avoiding the cost of TCP/IP communication.

Applications that use the IBM MQ classes for Java to connect in bindings mode must run on the same system as the queue manager to which they are connecting.

The Java Runtime Environment, that is being used to run the IBM MQ classes for Java application, must be configured to load the IBM MQ classes for Java libraries; see “IBM MQ classes for Java libraries” on page 345 for further information.

For more information on how to write applications to use bindings mode connections, see “IBM MQ classes for Java connection modes” on page 359.

Related concepts

IBM MQ Java language interfaces

“Using IBM MQ classes for JMS/Jakarta Messaging” on page 79

IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

Related tasks

Tracing IBM MQ classes for Java applications

Troubleshooting Java and JMS problems

Why should I use IBM MQ classes for Java?

A Java application can use either IBM MQ classes for Java or IBM MQ classes for JMS to access IBM MQ resources.

Note: Although existing applications that use the IBM MQ classes for Java continue to be fully supported, new applications should use the IBM MQ classes for Jakarta Messaging. Features that have recently been added to IBM MQ, such as asynchronous consume and automatic reconnection, are not available in the IBM MQ classes for Java, but are available in the IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging. For more information, see “Why should I use IBM MQ classes for JMS?” on page 81 and “Why should I use IBM MQ classes for Jakarta Messaging?” on page 80.

Note:

Stabilized IBM will make no further enhancements to the IBM MQ classes for Java and they are functionally stabilized at the level shipped in IBM MQ 8.0. Existing applications that use the IBM MQ classes for Java continue to be fully supported, but new features will not be added and requests for enhancements will be rejected. Fully supported means that defects will be fixed together with any changes necessitated by changes to IBM MQ System Requirements.

The IBM MQ classes for Java are not supported in IMS.

The IBM MQ classes for Java are not supported in WebSphere Liberty. They must not be used with either the IBM MQ Liberty messaging feature, or with the generic JCA support. For more information, see Using WebSphere MQ Java Interfaces in J2EE/JEE Environments.

Related concepts

“Accessing IBM MQ from Java - Choice of API” on page 82

IBM MQ provides three Java language interfaces.

Prerequisites for IBM MQ classes for Java

To use IBM MQ classes for Java, you need certain other software products.

For information about prerequisites for IBM MQ classes for Java, see the System Requirements for IBM MQ web page.

To develop IBM MQ classes for Java applications, you need a Java Development Kit (JDK). Details of the JDKs supported with your operating system can be found in the System Requirements for IBM MQ information.

To run IBM MQ classes for Java applications, you need the following software components:

- An IBM MQ queue manager, for applications that connect to a queue manager
- A Java Runtime Environment (JRE), for each system on which you run applications. A suitable JRE is supplied with IBM MQ.
- 

For IBM i, QShell, which is option 30 of the operating system
- 

For z/OS, z/OS UNIX System Services (z/OS UNIX)

If you require TLS connections to use cryptographic modules that have been FIPS 140-2 certified, you need the IBM Java JSSE FIPS provider (IBMJSSEFIPS). Every IBM JDK and JRE at version 1.4.2 or later contains IBMJSSEFIPS.

You can use Internet Protocol version 6 (IPv6) addresses in your IBM MQ classes for Java applications if IPv6 is supported by your Java virtual machine (JVM) and the TCP/IP implementation on your operating system.

Running IBM MQ classes for Java applications within Java EE

There are certain restrictions and design considerations that must be taken into account before using IBM MQ classes for Java in Java EE.

IBM MQ classes for Java has restrictions when used within a Java Platform, Enterprise Edition (Java EE) environment. There are also additional considerations that must be taken into account when designing, implementing, and managing an IBM MQ classes for Java application that runs inside a Java EE environment. These restrictions and considerations are outlined in the following sections.

JTA transactions restrictions

The only supported transaction manager for applications using IBM MQ classes for Java is IBM MQ itself. Although an application under JTA control can make use IBM MQ classes for Java, any work performed through these classes is not controlled by the JTA units of work. They instead form local units of work separate from those managed by the application server through the JTA interfaces. In particular, any rollback of the JTA transaction does not result in a rollback of any sent or received messages. This restriction applies to application or bean managed transactions and to container managed transactions, and all Java EE containers. To perform messaging work directly with IBM MQ inside application server-coordinated transactions, IBM MQ classes for JMS must be used instead.

Thread creation

IBM MQ classes for Java creates threads internally for various operations. For example, when running in BINDINGS mode to call directly on a local queue manager, the calls are made on a 'worker' thread created internally by IBM MQ classes for Java. Other threads can be created internally, for example to clear unused connections from a connection pool or to remove subscriptions for terminated publish/subscribe applications.

Some Java EE applications (for example those running in EJB and Web containers) must not create new threads. Instead, all work must be performed on the main application threads managed by the application

server. When applications use IBM MQ classes for Java, the application server might not be able to distinguish between application code and the IBM MQ classes for Java code, so the threads previously described cause the application to be non-compliant with the container specification. IBM MQ classes for JMS does not break these Java EE specifications and so can be used instead.

Security restrictions

Security policies implemented by an application server might prevent certain operations that are undertaken by the IBM MQ classes for Java API, such as creating and operating new threads of control (as described in the preceding sections).

For example, application servers typically run with Java Security disabled by default, and allow it to be enabled through some application server-specific configuration (some application servers also allow more detailed configuration of the policies used within Java Security). When Java Security is enabled, IBM MQ classes for Java might break the Java Security policy threading rules defined for the application server, and the API might not be able to create all the threads that it needs in order to function. To prevent problems with thread management, the use of IBM MQ classes for Java is not supported in environments where Java Security is enabled.

Application isolation considerations

An intended benefit of running applications within a Java EE environment is application isolation. The design and implementation of IBM MQ classes for Java predate the Java EE environment. IBM MQ classes for Java can be used in a manner which does not support the concept of application isolation. Specific examples of considerations in this area include:

- The use of static (JVM process-wide) settings within the MQEnvironment class, such as:
  - the user ID and password to be used for connection identification and authentication
  - the host name, port, and channel used for client connections
  - TLS configuration for secured client connections

  Modifying any of the MQEnvironment properties for the benefit of one application also affect other applications using the same properties. When running in a multi-application environment such as Java EE, each application must use its own distinct configuration through the creation of MQQueueManager objects with a specific set of properties, rather than defaulting to the properties configured in the process-wide MQEnvironment class.
- The MQEnvironment class introduces a number of static methods which act globally on all applications using IBM MQ classes for Java within the same JVM process, and there is no way to override this behavior for particular applications. Examples include:
  - configuring TLS properties, such as the location of the keystore
  - configuring client channel exits
  - enabling or disabling diagnostic tracing
  - managing the default connection pool used to optimize the use of connections to queue managers

  Invoking such methods affects all applications running in the same Java EE environment.
- Connection pooling is enabled to optimize the process of making multiple connections to the same queue manager. The default connection pool manager is process-wide, and shared by multiple applications. Changes to connection pool configuration, such as replacing the default connection manager for one application using the MQEnvironment.setDefaultConnectionManager() method therefore affects other applications running in the same Java EE application server.
- TLS is configured for applications using IBM MQ classes for Java using the MQEnvironment class and MQQueueManager object properties. It is not integrated with the managed security configuration of the application server itself. You must ensure that you configure IBM MQ classes for Java appropriately to provide your required level of security, and not use the application server configuration.

Bindings mode restrictions

IBM MQ and WebSphere Application Server can be installed on the same machine such that the major versions of the queue manager and of the IBM MQ resource adapter (RA) shipped in WebSphere Application Server are different.

If the queue manager and resource adapter major versions are different, bindings connections cannot be used. Any connections from WebSphere Application Server to the queue manager using the resource adapter must use client type connections. Bindings connections can be used if the versions are the same.

Character string conversions in IBM MQ classes for Java

The IBM MQ classes for Java use CharsetEncoders and CharsetDecoders directly for character string conversion. The default behavior for character string conversion can be configured with two system properties. The handling of messages that contain unmappable characters can be configured through `com.ibm.mq.MQMD`.

Before IBM MQ 8.0, string conversions in IBM MQ classes for Java was done by calling the `java.nio.charset.Charset.decode(ByteBuffer)` and `Charset.encode(CharBuffer)` methods.

Using either of these methods results in a default replacement (REPLACE) of malformed or untranslatable data. This behavior can obscure errors in applications, and lead to unexpected characters, for example ?, in translated data.

From IBM MQ 8.0, to detect such issues earlier and more effectively, the IBM MQ classes for Java use CharsetEncoders and CharsetDecoders directly and configure the handling of malformed and untranslatable data explicitly. The default behavior is to REPORT such issues by throwing a suitable `MQException`.

Configuring

Translating from UTF-16 (the character representation used in Java) to a native character set, such as UTF-8, is termed *encoding*, while translating in the opposite direction is termed *decoding*.

Decoding takes the default behavior for `CharsetDecoders`, reporting errors by throwing an exception.

One setting is used to specify a `java.nio.charset.CodingErrorAction` to control error handling on both encoding and decoding. One other setting is used to control the replacement byte, or bytes, when encoding. The default Java replacement String will be used in decoding operations.

Configuration of untranslatable data handling in IBM MQ classes for Java

From IBM MQ 8.0, `com.ibm.mq.MQMD` includes the following two fields:

byte[] unMappableReplacement
The byte sequence that will be written to an encoded string if an input character cannot be translated, and you have specified REPLACE.

Default: "?".getBytes()
The default Java replacement String is used in decoding operations.

java.nio.charset.CodingErrorAction unmappableAction
Specifies the action to be taken for untranslatable data on encoding and decoding:

Default: CodingErrorAction.REPORT;

System properties for setting system defaults

From IBM MQ 8.0, the following two Java system properties are available to configure default behavior regarding character string conversion.

com.ibm.mq.cfg.jmqi.UnmappableCharacterAction
Specifies the action to be taken for untranslatable data on encoding and decoding. The value can be REPORT, REPLACE, or IGNORE.

com.ibm.mq.cfg.jmqi.UnmappableCharacterReplacement
Sets or gets the replacement bytes to apply when a character cannot be mapped in an encoding operation The default Java replacement string is used in decoding operations.

To avoid confusion between Java character and native byte representations, you should specify `com.ibm.mq.cfg.jmqi.UnmappableCharacterReplacement` as a decimal number representing the replacement byte in the native character set.

For example, the decimal value of ?, as a native byte, is 63 if the native character set is ASCII-based, such as ISO-8859-1, while it is 111 if the native character set is EBCDIC.

Note: Note that if an MQMD or MQMessage object has either the **`unmappableAction`** or **`unMappableReplacement`** fields set, then the values of these fields take precedence over the Java system properties. This allows the values specified by the Java system properties to be overridden for each message if required.

Related concepts
"Character string conversions in IBM MQ classes for JMS" on page 133
The IBM MQ classes for JMS use CharsetEncoders and CharsetDecoders directly for character string conversion. The default behavior for character string conversion can be configured with two system properties. The handling of messages that contain unmappable characters can be configured through message properties for setting the UnmappableCharacterAction and the replacement bytes.

Installing and configuring IBM MQ classes for Java

This section describes the directories and files that are created when you install IBM MQ classes for Java, and tells you how to configure IBM MQ classes for Java after installation.

What is installed for IBM MQ classes for Java

The latest version of IBM MQ classes for Java is installed with IBM MQ. You might need to override default installation options to make sure this is done.

For more information about installing IBM MQ see:

- Multi Installing IBM MQ
- z/OS Installing the IBM MQ for z/OS product

IBM MQ classes for Java are contained in the Java archive (JAR) files, `com.ibm.mq.jar`, and `com.ibm.mq.jmqi.jar`.

Support for standard message headers, such as Programmable Command Format (PCF), is contained in the JAR file `com.ibm.mq.headers.jar`.

Support for Programmable Command Format (PCF) is contained in the JAR file `com.ibm.mq.pcf.jar`.

Note: It is not recommended to use the IBM MQ classes for Java within an application server. For information about the restrictions that apply when running in this environment, see "Running IBM MQ classes for Java applications within Java EE" on page 337. For more information, see Using WebSphere MQ Java Interfaces in J2EE/JEE Environments.

Important: Apart from the relocatable JAR files described in "IBM MQ classes for Java relocatable JAR files" on page 340, copying the IBM MQ classes for Java JAR files or native libraries to other machines, or to a different location on a machine where the IBM MQ classes for Java have been installed, is not supported.

IBM MQ classes for Java relocatable JAR files
The relocatable JAR files can be moved to systems that need to run IBM MQ classes for Java.

Important:

- Apart from the relocatable JAR files described in Relocatable JAR files, copying the IBM MQ classes for Java JAR files or native libraries to other machines, or to a different location on a machine where the IBM MQ classes for Java have been installed, is not supported.
- To avoid classloader conflicts, it is not recommended to bundle the relocatable JAR files within multiple applications inside the same Java runtime. In this scenario, consider making the IBM MQ relocatable JAR files available on the Java runtime's classpath.
- Do not include the relocatable JAR files within applications deployed into Java EE application servers, such as WebSphere Application Server. In these environments, the IBM MQ resource adapter should be deployed and used instead, as this contains the IBM MQ classes for Java. Note that WebSphere Application Server embeds the IBM MQ resource adapter, so there is no need to deploy it manually into this environment. In addition to this, the IBM MQ classes for Java are not supported in WebSphere Liberty. For more information, see "Liberty and the IBM MQ resource adapter" on page 425.
- If you are bundling the relocatable JAR files within your applications, ensure that you include all prerequisite JAR files as described in Relocatable JAR files. You should also ensure that you have appropriate procedures to update the bundled JAR files as part of application maintenance, to ensure that the IBM MQ classes for Java remain current and known issues are re-mediated.

Relocatable JAR files

Within an enterprise, the following files can be moved to systems that need to run IBM MQ classes for Java applications:

- JMS 2.0 `com.ibm.mq.allclient.jar` "1" on page 341
- JMS 3.0 `com.ibm.mq.jakarta.client.jar` "2" on page 341
- V 9.4.0 `bcpkix-jdk18on.jar` "3" on page 341
- `bcpkix-jdk15to18.jar` "4" on page 341
- V 9.4.0 `bcprov-jdk18on.jar` "3" on page 341
- `bcprov-jdk15to18.jar` "4" on page 341
- V 9.4.0 `bcutil-jdk18on.jar` "3" on page 341
- `bcutil-jdk15to18.jar` "4" on page 341
- `org.json.jar`

Notes:

1. JMS 2.0 and JMS 1.1
2. Jakarta Messaging 3.0
3. From IBM MQ 9.4.0
4. Before IBM MQ 9.4.0

Bouncy Castle security provider and CMS support JAR files

The Bouncy Castle security provider and CMS support JAR files are required. For more information, see Support for non-IBM JREs with AMS.

V 9.4.0 The following JAR files are required:

- `bcpkix-jdk18on.jar`
- `bcprov-jdk18on.jar`
- `bcutil-jdk18on.jar`

org.json.jar

The `org.json.jar` file is required if your IBM MQ classes for Java application uses a CCDT in JSON format.

com.ibm.mq.allclient.jar and com.ibm.mq.jakarta.client.jar

The files `com.ibm.mq.allclient.jar` and `com.ibm.mq.jakarta.client.jar` contain the IBM MQ classes for JMS, the IBM MQ classes for Java, and the PCF and Headers Classes. If you move these files to a new location, make sure that you take steps to keep this new location maintained with new IBM MQ Fix Packs. Also, make sure that the use of the files is made known to IBM Support if you are getting an interim fix.

To determine the version of the `com.ibm.mq.allclient.jar` file or the `com.ibm.mq.jakarta.client.jar` file, use the following command:

JMS 3.0

```
java -jar com.ibm.mq.jakarta.client.jar
```

JMS 2.0

```
java -jar com.ibm.mq.allclient.jar
```

The following example shows some sample output from this command:

```
C:\Program Files\IBM\MQ_1\java\lib>java -jar com.ibm.mq.allclient.jar
Name:       Java Message Service Client
Version:    9.3.0.0
Level:      p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar

Name:       IBM MQ classes for Java Message Service
Version:    9.3.0.0
Level:      p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar

Name:       IBM MQ JMS Provider
Version:    9.3.0.0
Level:      p000-L140428.1 mqjbnd=p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar

Name:       Common Services for Java Platform, Standard Edition
Version:    9.3.0.0
Level:      p000-L140428.1
Build Type: Production
Location:   file:/C:/Program Files/IBM/MQ_1/java/lib/com.ibm.mq.allclient.jar
```

Installation directories for IBM MQ classes for Java
IBM MQ classes for Java files and samples are installed in different locations according to platform. The location of the Java Runtime Environment (JRE) that is installed with IBM MQ also varies according to the platform.

Installation directories for IBM MQ classes for Java files

Table 49 on page 342 shows where the IBM MQ classes for Java files are installed.

Table 49. IBM MQ classes for Java installation directories

Platform	Directory
AIX AIX	*MQ_INSTALLATION_PATH*/java/lib
	/QIBM/ProdData/mqm/java/lib

Table 49. IBM MQ classes for Java installation directories (continued)	
Platform	**Directory**
Linux	*MQ_INSTALLATION_PATH*/java/lib
Windows	*MQ_INSTALLATION_PATH*\java\lib
z/OS	*MQ_INSTALLATION_PATH*/mqm/V8R0M0/java /lib

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Installation directories for samples

Some sample applications, such as the Installation Verification Programs (IVPs), are supplied with IBM MQ. Table 50 on page 343 shows where the sample applications are installed. The IBM MQ classes for Java samples are in a subdirectory called `wmqjava`. The PCF samples are in a subdirectory called `pcf`.

Table 50. Samples directories	
Platform	**Directory**
AIX	*MQ_INSTALLATION_PATH*/samp/wmqjava/
IBM i	/QIBM/ProdData/mqm/java/samples
Linux	*MQ_INSTALLATION_PATH*/samp/wmqjava/
Windows	*MQ_INSTALLATION_PATH*\tools\wmqjava\
z/OS	*MQ_INSTALLATION_PATH*/mqm/V8R0M0/java/samples

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Installation directories for JRE

The IBM MQ classes for JMS require a Java 7 (or above) Java Runtime Environment (JRE). A suitable JRE is installed with IBM MQ. Table 51 on page 343 shows where this JRE is installed. To run Java programs such as the provided samples, using this JRE, either explicitly invoke *JRE_LOCATION*`/bin/java` or add *JRE_LOCATION*`/bin` to the PATH environment (or equivalent) for your platform, where *JRE_LOCATION* is the directory given in Table 51 on page 343.

Table 51. JRE directories	
Platform	**Directory**
AIX	*MQ_INSTALLATION_PATH*/java/jre
IBM i	/QIBM/ProdData/mqm/java/jre
Linux	*MQ_INSTALLATION_PATH*/java/jre
Windows	*MQ_INSTALLATION_PATH*\java\jre
z/OS	*MQ_INSTALLATION_PATH*/mqm/V8R0M0/java/jre

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Environment variables relevant to IBM MQ classes for Java
If you want to run IBM MQ classes for Java applications, their class path must include the IBM MQ classes for Java and samples directories.

For IBM MQ classes for Java applications to run, their class path must include the appropriate IBM MQ classes for Java directory. To run the sample applications, the class path must also include the appropriate samples directories. This information can be provided in the Java invocation command or in the **CLASSPATH** environment variable.

Important: Setting the Java option `-Xbootclasspath` to include the IBM MQ classes for Java is not supported.

Table 52 on page 344 shows the appropriate **CLASSPATH** setting to use on each platform to run IBM MQ classes for Java applications, including the sample applications.

*Table 52. **CLASSPATH** setting to run IBM MQ classes for Java applications, including the IBM MQ classes for Java sample applications*

Platform	CLASSPATH setting
AIX	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.jar: *MQ_INSTALLATION_PATH*/samp/wmqjava/samples:
IBM i	CLASSPATH=/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar: /QIBM/ProdData/mqm/java/samples/wmqjava/samples:
Linux	CLASSPATH= *MQ_INSTALLATION_PATH*/java/lib/com.ibm.mq.jar: *MQ_INSTALLATION_PATH*/samp/wmqjava/samples:
Windows	CLASSPATH= *MQ_INSTALLATION_PATH*\Java\lib\com.ibm.mq.jar; *MQ_INSTALLATION_PATH*\tools\wmqjava\samples;
z/OS	CLASSPATH= *MQ_INSTALLATION_PATH*/mqm/V9R4M0/java/lib/com.ibm.mq.jar: *MQ_INSTALLATION_PATH*/mqm/V9R4M0/java/samples/wmqjava: *MQ_INSTALLATION_PATH*/mqm/V9R4M0/java/samples/pcf

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

If you compile using the `-Xlint` option, you might see a message warning you that `com.ibm.mq.ese.jar` is not present. You can ignore the warning. This file is only present if you have installed Advanced Message Security.

The scripts provided with IBM MQ classes for JMS use the following environment variables:

MQ_JAVA_DATA_PATH
This environment variable specifies the directory for log and trace output.

MQ_JAVA_INSTALL_PATH
This environment variable specifies the directory where IBM MQ classes for Java are installed, as shown in IBM MQ classes for Java installation directories.

MQ_JAVA_LIB_PATH
This environment variable specifies the directory where the IBM MQ classes for Java libraries are stored, as shown in The location of the IBM MQ classes for Java libraries for each platform. Some scripts supplied with IBM MQ classes for Java, such as IVTRun, use this environment variable.

Windows On Windows, all the environment variables are set automatically during installation.

Linux AIX On AIX and Linux, you can use the script `setjmsenv` (if you are using a 32-bit JVM) or `setjmsenv64` (if you are using a 64-bit JVM) to set the environment variables. These scripts are in the *MQ_INSTALLATION_PATH*/java/bin directory.

IBM i On IBM i, the environment variable **QIBM_MULTI_THREADED** must be set to Y. You can then run multithreaded applications in the same way that you run single threaded applications. For more information, see Setting up IBM MQ with Java and JMS.

IBM MQ classes for Java require a Java 7 Java Runtime Environment (JRE). For information about the location of a suitable JRE that is installed with IBM MQ, see "Installation directories for IBM MQ classes for Java" on page 342.

IBM MQ classes for Java libraries
The location of the IBM MQ classes for Java libraries varies according to platform. Specify this location when you start an application.

To specify the location of the Java Native Interface (JNI) libraries, start your application by using a **java** command with the following format:

```
java -Djava.library.path= library_path application_name
```

where *library_path* is the path to the IBM MQ classes for Java, which include the JNI libraries. Table 53 on page 345 shows the location of the IBM MQ classes for Java libraries for each platform. In this table, *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

Table 53. The location of the IBM MQ classes for Java libraries for each platform

Platform	**Directory containing the IBM MQ classes for Java libraries**
AIX AIX	*MQ_INSTALLATION_PATH*/java/lib (32-bit libraries) *MQ_INSTALLATION_PATH*/java/lib64 (64-bit libraries)
Linux Linux (x86 platform)	*MQ_INSTALLATION_PATH*/java/lib
Linux Linux (POWER, x86-64 and zSeries s390x platforms)	*MQ_INSTALLATION_PATH*/java/lib (32-bit libraries) *MQ_INSTALLATION_PATH*/java/lib64 (64-bit libraries)
Windows Windows	*MQ_INSTALLATION_PATH*\Java\lib (32-bit libraries) *MQ_INSTALLATION_PATH*\Java\lib64 (64-bit libraries)
z/OS z/OS	*MQ_INSTALLATION_PATH*/mqm/V8R0M0/java/lib (32-bit and 64-bit libraries)

Note:

1. Linux AIX On AIX or Linux (Power platform), use either the 32-bit libraries or the 64-bit libraries. Use the 64-bit libraries only if you are running your application in a 64-bit Java virtual machine (JVM) on a 64-bit platform. Otherwise, use the 32-bit libraries.
2. Windows On Windows, you can use the PATH environment variable to specify the location of the IBM MQ classes for Java libraries instead of specifying their location on the **java** command.
3. IBM i To use IBM MQ classes for Java in bindings mode on IBM i, ensure that the library QMQMJAVA is in your library list.

4. z/OS On z/OS, you can use either a 32-bit or 64-bit Java virtual machine (JVM). You do not have to specify which libraries to use; IBM MQ classes for Java can determine for itself which JNI libraries to load.

Related concepts

Using IBM MQ classes for Java
After installing IBM MQ classes for Java, you can configure your installation so that you run your own applications.

Support for OSGi with IBM MQ classes for Java
OSGi provides a framework that supports the deployment of applications as bundles. Three OSGi bundles are supplied as part of the IBM MQ classes for Java.

OSGi provides a general purpose, secure, and managed Java framework, which supports the deployment of applications that come in the form of bundles. OSGi-compliant devices can download and install bundles, and remove them when they are no longer required. The framework manages the installation and updating of bundles in a dynamic and scalable fashion.

The IBM MQ classes for Java include the following OSGi bundles.

com.ibm.mq.osgi.java_*version_number*.jar
The JAR files to allow applications to use the IBM MQ classes for Java.

com.ibm.mq.jakarta.osgi.allclient_*version_number*.jar
JMS 3.0 For Jakarta Messaging 3.0, this JAR file allows applications to use both the IBM MQ classes for JMS and the IBM MQ classes for Java, and also includes the code to handle PCF messages.

com.ibm.mq.osgi.allclient_*version_number*.jar
JMS 2.0 For JMS 2.0, this JAR file allows applications to use both the IBM MQ classes for JMS and the IBM MQ classes for Java, and also includes the code to handle PCF messages.

com.ibm.mq.jakarta.osgi.allclientprereqs_*version_number*.jar
JMS 3.0 For Jakarta Messaging 3.0, this JAR file provides the prerequisites for `com.ibm.mq.jakarta.osgi.allclient_version_number.jar`.

com.ibm.mq.osgi.allclientprereqs_*version_number*.jar
JMS 2.0 For JMS 2.0, this JAR file provides the prerequisites for `com.ibm.mq.osgi.allclient_version_number.jar`.

where *version_number* is the version number of IBM MQ that is installed.

The bundles are installed into the `java/lib/OSGi` subdirectory of your IBM MQ installation, or the `java\lib\OSGi` folder on Windows.

From IBM MQ 8.0, use the bundles `com.ibm.mq.osgi.allclient_8.0.0.0.jar`, and `com.ibm.mq.osgi.allclientprereqs_8.0.0.0.jar` for any new applications. Using these bundles removes the restriction of not being able to run both IBM MQ classes for JMS and the IBM MQ classes for Java within the same OSGi framework. All other restrictions still apply however. For versions of IBM MQ before IBM MQ 8.0, the restriction of using either IBM MQ classes for JMS or IBM MQ classes for Java applies.

Nine other bundles are also installed into the `java/lib/OSGi` sub directory of your IBM MQ installation, or the `java\lib\OSGi` folder on Windows. These bundles are part of the IBM MQ classes for JMS, and must not be loaded into an OSGi runtime environment that has the IBM MQ classes for Java bundle loaded. If the IBM MQ classes for Java OSGi bundle is loaded into an OSGi runtime environment that also has the IBM MQ classes for JMS bundles loaded, errors as shown in the following example occur when applications using either the IBM MQ classes for Java bundle or the IBM MQ classes for JMS bundles are run:

```
java.lang.ClassCastException: com.ibm.mq.MQException incompatible with com.ibm.mq.MQException
```

The OSGi bundle for the IBM MQ classes for Java has been written to the OSGi Release 4 specification; it does not work in an OSGi Release 3 environment.

You must set your system path or library path correctly so that the OSGi runtime environment can find any required DLL files or shared libraries.

If you use the OSGi bundle for the IBM MQ classes for Java, channel exit classes written in Java are not supported because of an inherent problem in loading classes in a multiple class loader environment such as OSGi. A user bundle can be aware of the IBM MQ classes for Java bundle, but the IBM MQ classes for Java bundle is not aware of any user bundle. As a result, the class loader used in an IBM MQ classes for Java bundle cannot load a channel exit class that is in a user bundle.

For more information about OSGi, see the OSGi alliance website.

z/OS *Installation of IBM MQ classes for Java on z/OS*

On z/OS, the STEPLIB used at runtime must contain the IBM MQ SCSQAUTH and SCSQANLE libraries.

From z/OS UNIX System Services, you can add these libraries by using a line in your `.profile` as shown in the following example, replacing `thlqual` with the high level data set qualifier that you chose when installing IBM MQ:

```
export STEPLIB=thlqual.SCSQAUTH:thlqual.SCSQANLE:$STEPLIB
```

In other environments, you typically need to edit the startup JCL to include SCSQAUTH on the STEPLIB concatenation:

```
 STEPLIB DD DSN=thlqual.SCSQAUTH,DISP=SHR
         DD DSN=thlqual.SCSQANLE,DISP=SHR
```

The IBM MQ classes for Java configuration file

An IBM MQ classes for Java configuration file specifies properties that are used to configure IBM MQ classes for Java.

The format of an IBM MQ classes for Java configuration file is that of a standard Java properties file.

A sample configuration file, `mqjava.config`, is supplied in the `bin` subdirectory of the IBM MQ classes for Java installation directory. This file documents all the supported properties and their default values.

Note: The sample configuration file is overwritten when the IBM MQ installation is upgraded to a future Fix Pack. Therefore, it is recommended that you make a copy of the sample configuration file for use with your applications.

You can choose the name and location of an IBM MQ classes for Java configuration file. When you start your application, use a **java** command with the following format:

```
java -Dcom.ibm.msg.client.config.location=config_file_url application_name
```

In the command, *config_file_url* is a uniform resource locator (URL) that specifies the name and location of the IBM MQ classes for Java configuration file. URLs of the following types are supported: `http`, `file`, `ftp`, and `jar`.

The following example shows a **java** command:

```
java -Dcom.ibm.msg.client.config.location=file:/D:/mydir/mqjava.config MyAppClass
```

This command identifies the IBM MQ classes for Java configuration file as the file `D:\mydir\mqjava.config` on the local Windows system.

When an application starts, IBM MQ classes for Java reads the contents of the configuration file and stores the specified properties in an internal property store. If the **java** command does not identify a configuration file, or if the configuration file cannot be found, IBM MQ classes for Java uses the default values for all the properties. If required, you can override any property in the configuration file by specifying it as a system property on the **java** command.

An IBM MQ classes for Java configuration file can be used with any of the supported transports between an application and a queue manager or broker.

Overriding properties specified in an IBM MQ MQI client configuration file

An IBM MQ MQI client configuration file can also specify properties that are used to configure IBM MQ classes for Java. However, properties that are specified in an IBM MQ MQI client configuration file apply only when an application connects to a queue manager in client mode.

If required, you can override any attribute in an IBM MQ MQI client configuration file by specifying it as a property in an IBM MQ classes for Java configuration file. To override an attribute in an IBM MQ MQI client configuration file, use an entry with the following format in the IBM MQ classes for Java configuration file:

```
com.ibm.mq.cfg.stanza.propName=propValue
```

The variables in the entry have the following meanings:

stanza
The name of the stanza in the IBM MQ MQI client configuration file that contains the attribute.

propName
The name of the attribute as specified in the IBM MQ MQI client configuration file.

propValue
The value of the property that overrides the value of the attribute that is specified in the IBM MQ MQI client configuration file.

Alternatively, you can override an attribute in an IBM MQ MQI client configuration file by specifying the property as a system property on the **java** command. Use the preceding format to specify the property as a system property.

Only the following attributes in an IBM MQ MQI client configuration file are relevant to IBM MQ classes for Java. If you specify or override other attributes, it has no effect. Specifically, note that the `ChannelDefinitionFile` and `ChannelDefinitionDirectory` in the CHANNELS stanza of the client configuration file are not used. See “Using a client channel definition table with IBM MQ classes for Java” on page 363 for details of how to use the CCDT with the IBM MQ classes for Java.

Table 54. Which stanza of the client configuration file contains which attribute

Stanza	Attribute
CHANNELS stanza of the client configuration file	Put1DefaultAlwaysSync
CHANNELS stanza of the client configuration file	PasswordProtection
ClientExitPath stanza of the client configuration file	ExitsDefaultPath
ClientExitPath stanza of the client configuration file	ExitsDefaultPath64
ClientExitPath stanza of the client configuration file	JavaExitsClasspath
JMQI stanza of the client configuration file	useMQCSPauthentication
MessageBuffer stanza of the client configuration file	MaximumSize
MessageBuffer stanza of the client configuration file	PurgeTime
MessageBuffer stanza of the client configuration file	UpdatePercentage
TCP stanza of the client configuration file	ClntRcvBuffSize
TCP stanza of the client configuration file	ClntSndBuffSize

Table 54. Which stanza of the client configuration file contains which attribute (continued)	
Stanza	**Attribute**
TCP stanza of the client configuration file	Connect_Timeout
TCP stanza of the client configuration file	KeepAlive

For more information on the IBM MQ MQI client configuration, see IBM MQ MQI client configuration file, `mqclient.ini`.

Related tasks

Tracing IBM MQ classes for Java applications

Using Java Standard Environment Trace to configure Java trace

Use the Java Standard Environment Trace Settings stanza to configure the IBM MQ classes for Java trace facility.

com.ibm.msg.client.commonservices.trace.outputName = *traceOutputName*

traceOutputName is the directory and file name to which trace output is sent.

By default, trace information is written to a trace file in the current working directory of the application. The name of the trace file depends upon the environment that the application is running in:

- If the application has loaded the IBM MQ classes for Java from the relocatable JAR file `com.ibm.mq.allclient.jar`, trace is written to a file called `mqjavaclient_%PID%.cl%u.trc`.
- If the application has loaded the IBM MQ classes for Java from the JAR file `com.ibm.mq.jar`, trace is written to a file called `mqjava_%PID%.cl%u.trc`.

where *%PID%* is the process identifier of the application that is being traced, and *%u* is a unique number to differentiate files between threads running trace under different Java classloaders.

If a process ID is unavailable, a random number is generated and prefixed with the letter `f`. To include the process ID in a file name you specify, use the string `%PID%`.

If you specify an alternative directory, it must exist, and you must have write permission for this directory. If you do not have write permission, the trace output is written to `System.err`.

com.ibm.msg.client.commonservices.trace.include = *includeList*

includeList is a list of packages and classes that are traced, or the special values `ALL` or `NONE`.

Separate package or class names with a semicolon, `;`. *includeList* defaults to `ALL`, and traces all packages and classes in IBM MQ classes for Java.

Note: You can include a package but then exclude subpackages of that package. For example, if you include package `a.b` and exclude package `a.b.x`, the trace includes everything in `a.b.y` and `a.b.z`, but not `a.b.x` or `a.b.x.1`.

com.ibm.msg.client.commonservices.trace.exclude = *excludeList*

excludeList is a list of packages and classes that are not traced, or the special values `ALL` or `NONE`.

Separate package or class names with a semicolon, `;`. *excludeList* defaults to `NONE`, and therefore excludes no packages and classes in IBM MQ classes for JMS from being traced.

Note: You can exclude a package but then include subpackages of that package. For example, if you exclude package `a.b` and include package `a.b.x`, the trace includes everything in `a.b.x` and `a.b.x.1`, but not `a.b.y` or `a.b.z`.

Any package or class that is specified, at the same level, as both included and excluded is included.

com.ibm.msg.client.commonservices.trace.maxBytes = *maxArrayBytes*

maxArrayBytes is the maximum number of bytes that are traced from any byte arrays.

If *maxArrayBytes* is set to a positive integer, it limits the number of bytes in a byte-array that are written out to the trace file. It truncates the byte array after writing *maxArrayBytes* out. Setting

maxArrayBytes reduces the size of the resulting trace file, and reduces the effect of tracing on the performance of the application.

A value of `0` for this property means that none of the contents of any byte arrays are sent to the trace file.

The default value is `-1`, which removes any limit on the number of bytes in a byte array that are sent to the trace file.

com.ibm.msg.client.commonservices.trace.limit = *maxTraceBytes*
maxTraceBytes is the maximum number of bytes that are written to a trace output file.

maxTraceBytes works with *traceCycles*. If the number of bytes of trace written is near to the limit, the file is closed, and a new trace output file is started.

A value of `0` means that a trace output file has zero length. The default value is `-1`, which means that the amount of data to be written to a trace output file is unlimited.

com.ibm.msg.client.commonservices.trace.count = *traceCycles*
traceCycles is the number of trace output files to cycle through.

If the current trace output file reaches the limit specified by *maxTraceBytes*, the file is closed. Further trace output is written to the next trace output file in sequence. Each trace output file is distinguished by a numeric suffix appended to the file name. The current or most recent trace output file is `mqjms.trc.0`, the next most recent trace output file is `mqjms.trc.1`. Older trace files follow the same numbering pattern up to the limit.

The default value of *traceCycles* is 1. If *traceCycles* is 1, when the current trace output file reaches its maximum size, the file is closed and deleted. A new trace output file with the same name is started. Therefore, only one trace output file exists at a time.

com.ibm.msg.client.commonservices.trace.parameter = *traceParameters*
traceParameters controls whether method parameters and return values are included in the trace.

traceParameters defaults to TRUE. If *traceParameters* is set to FALSE, only method signatures are traced.

com.ibm.msg.client.commonservices.trace.startup = *startup*
There is an initialization phase of IBM MQ classes for Java during which resources are allocated. The main trace facility is initialized during the resource allocation phase.

If *startup* is set to TRUE, startup trace is used. Trace information is produced immediately and includes the setup of all components, including the trace facility itself. Startup trace information can be used to diagnose configuration problems. Startup trace information is always written to `System.err`.

startup defaults to FALSE.

startup is checked before initialization is complete. For this reason, only specify the property on the command line as a Java system property. Do not specify it in the IBM MQ classes for Java configuration file.

com.ibm.msg.client.commonservices.trace.compress = *compressedTrace*
Set *compressedTrace* to TRUE to compress trace output.

The default value of *compressedTrace* is FALSE.

If *compressedTrace* is set to TRUE, trace output is compressed. The default trace output file name has the extension `.trz`. If compression is set to FALSE, the default value, the file has the extension `.trc` to indicate it is uncompressed. However if the file name for the trace output has been specified in *traceOutputName* that name is used instead; no suffix is applied to the file.

Compressed trace output is smaller than uncompressed. Because there is less I/O, it can be written out faster than uncompressed trace. Compressed tracing has less effect on the performance of IBM MQ classes for Java than uncompressed tracing.

If *maxTraceBytes* and *traceCycles* are set, multiple compressed trace files are created in place of multiple flat files.

If IBM MQ classes for Java ends in an uncontrolled manner, a compressed trace file might not be valid. For this reason, trace compression must only be used when IBM MQ classes for Java closes down in a controlled manner. Only use trace compression if the problems being investigated do not cause the JVM itself to stop unexpectedly. Do not use trace compression when diagnosing problems that can result in `System.Halt()` shutdowns or abnormal, uncontrolled JVM terminations.

com.ibm.msg.client.commonservices.trace.level = *traceLevel*

traceLevel specifies a filtering level for the trace. The defined trace levels are as follows:

- `TRACE_NONE`: `0`
- `TRACE_EXCEPTION`: `1`
- `TRACE_WARNING`: `3`
- `TRACE_INFO`: `6`
- `TRACE_ENTRYEXIT`: `8`
- `TRACE_DATA`: `9`
- `TRACE_ALL`: `Integer.MAX_VALUE`

Each trace level includes all lower levels. For example, if trace level is set at `TRACE_INFO`, then any trace point with a defined level of `TRACE_EXCEPTION`, `TRACE_WARNING`, or `TRACE_INFO` is written to the trace. All other trace points are excluded.

com.ibm.msg.client.commonservices.trace.standalone = *standaloneTrace*

standaloneTrace controls whether the IBM MQ classes for Java client tracing service is used in a WebSphere Application Server environment.

If *standaloneTrace* is set to TRUE, the IBM MQ classes for Java client tracing properties are used to determine the trace configuration.

If *standaloneTrace* is set to FALSE, and the IBM MQ classes for Java client is running in an WebSphere Application Server container, the WebSphere Application Server trace service is used. The trace information that is generated depends upon the trace settings of the application server.

The default value of *standaloneTrace* is FALSE.

IBM MQ classes for Java and software management tools

Software management tools such as Apache Maven can be used with the IBM MQ classes for Java.

Many large development organizations use these tools to centrally manage repositories of third-party libraries.

The IBM MQ classes for Java are composed of a number of JAR files. When you are developing Java language applications by using this API, an installation of either an IBM MQ Server, IBM MQ Client, or IBM MQ Client SupportPac is required on the machine where the application is being developed.

If you want to use a software management tool and add the JAR files that make up the IBM MQ classes for Java to a centrally managed repository, the following points must be observed:

- A repository or container must be made available only to developers within your organization. Any distribution outside of the organization is not permitted.
- The repository needs to contain a complete and consistent set of JAR files from a single IBM MQ release or Fix Pack.
- You are responsible for updating the repository with any maintenance provided by IBM Support.

From IBM MQ 8.0, the `com.ibm.mq.allclient.jar` JAR file needs to be installed into the repository.

From IBM MQ 9.0, the Bouncy Castle security provider and CMS support JAR files are required. For more information, see “IBM MQ classes for Java relocatable JAR files” on page 340 and Support for non-IBM JREs.

Post installation setup for IBM MQ classes for Java applications

After installing IBM MQ classes for Java, you can configure your installation so that you run your own applications.

Remember to check the IBM MQ product readme file for the latest information, or for more specific information about your environment. The latest version of the product readme file is available on the IBM MQ, WebSphere MQ, and MQSeries® product readmes web page.

Before attempting to run an IBM MQ classes for Java application in bindings mode, make sure that you have configured IBM MQ as described in Configuring.

Configuring your queue manager to accept client connections from IBM MQ classes for Java

To configure your queue manager to accept incoming connection requests from clients, define and permit use of a server connection channel and start a listener program.

See "Configuring a queue manager to accept client connections on Multiplatforms" on page 1028 for details.

Running IBM MQ classes for Java applications under the Java security manager

IBM MQ classes for Java can run with the Java security manager enabled. To successfully run applications with the Java security manager enabled, you must configure your Java Virtual Machine (JVM) with a suitable policy definition file.

The simplest way to create a suitable policy definition file is to change the policy file supplied with the Java runtime environment (JRE). On most systems, this file is stored in the `path lib/security/java.policy`, relative to your JRE directory. You can edit policy files either by using your preferred editor or by using the policytool program supplied with your JRE.

You must give authority to the `com.ibm.mq.jmqi.jar` file so that it can:

- Create sockets (in client mode)
- Load the native library (in bindings mode)
- Read various properties from the environment

The system property **`os.name`** must be available to the IBM MQ classes for Java when running under the Java security manager.

If your Java application uses the Java security manager, you must add the following permission to the `java.security.policy` file used by the application, otherwise, exceptions will be thrown to the application:

```
permission java.lang.RuntimePermission "modifyThread";
```

This RuntimePermission is required by the client as part of managing the assignment and closure of multiplexed conversations over TCP/IP connections to queue managers.

Example policy file entry

Here is an example of a policy file entry that allows IBM MQ classes for Java to run successfully under the default security manager. Replace the string *MQ_INSTALLATION_PATH* in this example with the location where IBM MQ classes for Java are installed on your system.

```
grant codeBase "file: MQ_INSTALLATION_PATH/java/lib/*" {
//We need access to these properties, mainly for tracing
permission java.util.PropertyPermission "user.name","read";
permission java.util.PropertyPermission "os.name","read";
permission java.util.PropertyPermission "user.dir","read";
permission java.util.PropertyPermission "line.separator","read";
permission java.util.PropertyPermission "path.separator","read";
permission java.util.PropertyPermission "file.separator","read";
permission java.util.PropertyPermission "com.ibm.msg.client.commonservices.log.*","read";
permission java.util.PropertyPermission "com.ibm.msg.client.commonservices.trace.*","read";
permission java.util.PropertyPermission "Diagnostics.Java.Errors.Destination.Filename","read";
permission java.util.PropertyPermission "com.ibm.mq.commonservices","read";
permission java.util.PropertyPermission "com.ibm.mq.cfg.*","read";
```

```
//Tracing - we need the ability to control java.util.logging
permission java.util.logging.LoggingPermission "control";
// And access to create the trace file and read the log file - assumed to be in the current
directory
permission java.io.FilePermission "*","read,write";

// Required to allow a trace file to be written to the filesystem.
// Replace 'TRACE_FILE_DIRECTORY' with the directory name where trace is to be written to
permission java.io.FilePermission "TRACE_FILE_DIRECTORY","read,write";
permission java.io.FilePermission "TRACE_FILE_DIRECTORY/*","read,write";

// We'd like to set up an mBean to control trace
permission javax.management.MBeanServerPermission "createMBeanServer";
permission javax.management.MBeanPermission "*","*";

// We need to be able to read manifests etc from the jar files in the installation directory
permission java.io.FilePermission "MQ_INSTALLATION_PATH/java/lib/-","read";

//Required if mqclient.ini/mqs.ini configuration files are used
permission java.io.FilePermission "MQ_DATA_DIRECTORY/mqclient.ini","read";
permission java.io.FilePermission "MQ_DATA_DIRECTORY/mqs.ini","read";

//For the client transport type.
permission java.net.SocketPermission "*","connect,resolve";

//For the bindings transport type.
permission java.lang.RuntimePermission "loadLibrary.*";

//For applications that use CCDT tables (access to the CCDT AMQCLCHL.TAB)
permission java.io.FilePermission "MQ_DATA_DIRECTORY/qmgrs/QM_NAME/@ipcc/AMQCLCHL.TAB","read";

//For applications that use User Exits
permission java.io.FilePermission "MQ_DATA_DIRECTORY/exits/*","read";
permission java.io.FilePermission "MQ_DATA_DIRECTORY/exits64/*","read";
permission java.lang.RuntimePermission "createClassLoader";

//Required for the z/OS platform
permission java.util.PropertyPermission "com.ibm.vm.bitmode","read";

// Used by the internal ConnectionFactory implementation
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";

// Used for controlled class loading
permission java.lang.RuntimePermission "setContextClassLoader";

// Used to default the Application name in Client mode connections
permission java.util.PropertyPermission "sun.java.command","read";

// Used by the IBM JSSE classes
permission java.util.PropertyPermission "com.ibm.crypto.provider.AESNITrace","read";

//Required to determine if an IBM Java Runtime is running in FIPS mode,
//and to modify the property values status as required.
permission java.util.PropertyPermission "com.ibm.jsse2.usefipsprovider","read,write";
permission java.util.PropertyPermission "com.ibm.jsse2.JSSEFIPS","read,write";
//Required if an IBM FIPS provider is to be used for SSL communication.
permission java.security.SecurityPermission "insertProvider.IBMJCEFIPS";

// Required for non-IBM Java Runtimes that establish secure client
// transport mode connections using mutual TLS authentication
permission java.util.PropertyPermission "javax.net.ssl.keyStore","read";
permission java.util.PropertyPermission "javax.net.ssl.keyStorePassword","read";

// Required for Java applications that use the Java Security Manager
permission java.lang.RuntimePermission "modifyThread";
};
```

This example of a policy file enables the IBM MQ classes for Java to work correctly under the security manager, but you might still need to enable your own code to run correctly before your applications work.

The sample code shipped with IBM MQ classes for Java has not been specifically enabled for use with the security manager; however the IVT tests run with this policy file and the default security manager in place.

Important:

The IBM MQ classes for Java trace facility requires further permissions as it performs additional querying of system properties, and also further file system operations.

A suitable template security policy file for running under a security manager with tracing enabled is provided in the `samples/wmqjava` directory of the IBM MQ installation as `example.security.policy`.

For a default installation, the `example.security.policy` file is located, on:

Windows
In `C:\Program Files\IBM\MQ\Tools\wmqjava\samples\example.security.policy`

Linux
In `/opt/mqm/samp/wmqjava/samples/example.security.policy`

Solaris
In `/opt/mqm/samp/wmqjava/samples/example.security.policy`

AIX
In `/usr/mqm/samp/wmqjava/samples/example.security.policy`

Running IBM MQ classes for Java applications under CICS Transaction Server
An IBM MQ classes for Java application can be run as a transaction under CICS Transaction Server.

To run an IBM MQ classes for Java application as a transaction under CICS Transaction Server for z/OS, perform the following steps:

1. Define the application and transaction to CICS by using the supplied CEDA transaction.
2. Ensure that the IBM MQ CICS adapter is installed in your CICS system. z/OS (See Using IBM MQ with CICS for details.)
3. Ensure that the JVM environment specified in CICS includes the appropriate CLASSPATH and LIBPATH entries.
4. Initiate the transaction by using any of your normal processes.

For more information on running CICS Java transactions, refer to your CICS system documentation.

Verifying the IBM MQ classes for Java installation

An installation verification program, MQIVP, is supplied with IBM MQ classes for Java. You can use this program to test all the connection modes of IBM MQ classes for Java.

The program prompts for a number of choices and other data to determine which connection mode you want to verify. Use the following procedure to verify your installation:

1. If you are going to run the program in client mode, configure your queue manager as described in “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028. The queue to use is SYSTEM.DEFAULT.LOCAL.QUEUE
2. If you are going to run the program in client mode, see also “Using IBM MQ classes for Java” on page 335.

   Perform the remaining steps of this procedure on the system on which you are going to run the program.
3. Make sure that you have updated your CLASSPATH environment variable according to the instructions in “Environment variables relevant to IBM MQ classes for Java” on page 344.
4. Change Directory to *MQ_INSTALLATION_PATH*`/mqm/samp/wmqjava/samples`, where *MQ_INSTALLATION_PATH* is the path to your IBM MQ installation. Then at the command prompt, enter:

   ```
   java -Djava.library.path= library_path MQIVP
   ```

   where *library_path* is the path to the IBM MQ classes for Java libraries (see “IBM MQ classes for Java libraries” on page 345).

   At the prompt marked (1):

   - To use a TCP/IP connection, enter an IBM MQ server host name.
   - To use native connection (bindings mode), leave the field blank (do not enter a name).

The program tries to:

- 1. Connect to the queue manager
- 2. Open the queue SYSTEM.DEFAULT.LOCAL.QUEUE, put a message on the queue, get a message from the queue, and then close the queue
- 3. Disconnect from the queue manager
- 4. Return a message if the operations are successful

Here is an example of the prompts and responses you might see. The actual prompts and your responses depend on your IBM MQ network.

```
Please enter the IP address of the MQ server    : ipaddress(1)
Please enter the port to connect to             : (1414) (2)
Please enter the server connection channel name : channelname (2)
Please enter the queue manager name             : qmname
Success: Connected to queue manager.
Success: Opened SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Put a message to SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Got a message from SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Closed SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Disconnected from queue manager


Tests complete -
SUCCESS: This MQ Transport is functioning correctly.
Press Enter to continue ...
```

Note:

1. z/OS On z/OS, leave the field blank at prompt marked (1).
2. If you choose server connection, you do not see the prompts marked (2).
3. IBM i On IBM i, you can only issue the `java MQIVP` command from QShell. Alternatively, you can run the application by using the CL command `RUNJVA CLASS(MQIVP)`.

Using the IBM MQ classes for Java sample applications

The IBM MQ classes for Java sample applications provide an overview of the common features of IBM MQ classes for Java API. You can use them to verify your installation and messaging server set up and to help you build your own applications.

About this task

If you need help to create your own applications, you can use the sample applications as a starting point. Both the source and a compiled version are provided for each application. Review the sample source code and identify the key steps to create each required object for your application (MQQueueManager, MQConstants , MQMessage, MQPutMessageOptions and MQDestination), and to set any specific properties that are needed to specify how you want your application to work. For more information, see “Writing IBM MQ classes for Java applications” on page 359. The samples might be subject to change in future releases of IBM MQ Java.

Table 55 on page 355 shows where the IBM MQ classes for Java sample applications are installed on each platform:

Table 55. Installation directories for the IBM MQ classes for Java sample applications

Platform	Directory
AIX AIX Linux Linux	*MQ_INSTALLATION_PATH*`/samp/wmqjava/samples`
Windows Windows	*MQ_INSTALLATION_PATH*`\tools\wmqjava\samples`

Table 55. Installation directories for the IBM MQ classes for Java sample applications (continued)	
Platform	**Directory**
IBM i	`/qibm/proddata/mqm/java/samples/wmqjava/samples`
z/OS	*MQ_INSTALLATION_PATH*`/java/samples/wmqjava`

Table 56 on page 356 shows the sets of sample applications that are supplied with IBM MQ classes for Java.

Table 56. IBM MQ classes for Java sample applications	
Name of sample	**Description**
IMSBridgeSample.java	Simple program to demonstrate using the IMS Bridge with the IBM MQ classes for Java.
MQIVP.java	IBM MQ Java installation verification program.
MQMessagePropertiesSample.java	Demonstrates the use of the Message Properties API .
MQPubSubApiSample.java	Demonstrates using the publish/subscribe API.
MQSample.java	Simple program to demonstrate putting and getting a message from a queue.
MQSampleMessageManager.java	Utility class for message handling in the IBM MQ base Java samples.
mqjcivp.properties	This resource bundle contains the messages used by the IBM MQ classes for Java installation verification program (`MQIVP.java`).

The IBM MQ classes for Java provide a script called `runjms` that can be used to run the sample applications. This script sets up the IBM MQ environment to allow you to run the IBM MQ classes for Java sample applications.

Table 57 on page 356 shows the location of the script on each platform:

Table 57. Location of the `runjms` script	
Platform	**Directory**
AIX Linux	*MQ_INSTALLATION_PATH*`/java/bin/runjms`
Windows	*MQ_INSTALLATION_PATH*`\java\bin\runjms.bat`

IBM i	`/qibm/proddata/mqm/java/bin/runjms`
	or
	`/qibm/proddata/mqm/java/bin/runjms64`

z/OS	*MQ_INSTALLATION_PATH*`java/bin/runjms`

To use the `runjms` script to invoke a sample application, complete the following steps:

Procedure

1. Bring up a command prompt and navigate to the directory containing the sample application that you want to run.
2. Enter the following command:

   ```
   Path to the runjms script/runjms sample_application_name
   ```

   The sample application displays a list of parameters that it needs.
3. Enter the following command to run the sample with these parameters:

   ```
   Path to the runjms script/runjms sample_application_name parameters
   ```

Example

Linux For example, to run the MQIVP sample on Linux, enter the following commands:

```
cd /opt/mqm/samp/wmqjava/samples
/opt/mqm/java/bin/runjms MQIVP
```

Related concepts

"What is installed for IBM MQ classes for JMS" on page 85
A number of files and directories are created when you install IBM MQ classes for JMS. On Windows, some configuration is performed during installation by automatically setting environment variables. On other platforms, and in certain Windows environments, you must set environment variables before you can run IBM MQ classes for JMS applications.

Solving IBM MQ classes for Java problems

Initially, run the installation verification program. You might also have to use the trace facility.

If an application does not complete successfully, run the installation verification program, and follow the advice given in the diagnostic messages. The installation verification program is described in "Verifying the IBM MQ classes for Java installation" on page 354.

If the problems continue and you need to contact the IBM service team, you might be asked to turn on the trace facility. Do this as shown in the following example.

To trace the `MQIVP` program:

- Create a `com.ibm.mq.commonservices` properties file (see Using com.ibm.mq.commonservices.
- Enter the following command:

```
java -Dcom.ibm.mq.commonservices=commonservices_properties_file java
-Djava.library.path= library_path MQIVP -trace
```

where:

- *commonservices_properties_file* is the path (including the filename) to the `com.ibm.mq.commonservices` properties file.
- *library_path* is the path to the IBM MQ classes for Java libraries (see "IBM MQ classes for Java libraries" on page 345).

For more information about how to use trace, see Tracing IBM MQ classes for Java applications.

z/OS MQ Adv. VUE Java client connectivity to batch applications running on z/OS

Under certain conditions, an IBM MQ classes for Java application on z/OS can connect to a queue manager on z/OS by using a client connection. Use of a client connection can simplify IBM MQ topologies.

By using a client connection, an IBM MQ classes for Java application can connect to a remote z/OS queue manager if the following conditions apply:

- The application is running in a batch environment.
- The queue manager being connected to is running with IBM MQ Advanced for z/OS Value Unit Edition entitlement, and therefore has the **ADVCAP** parameter set to ENABLED.

For more information on IBM MQ Advanced for z/OS Value Unit Edition see IBM MQ product identifiers and export information.

See DISPLAY QMGR for more information on **ADVCAP** and START QMGR for more information on **QMGRPROD**.

An IBM MQ classes for Java application on z/OS cannot use a client mode connection to connect to a queue manager that is not running on z/OS

If an IBM MQ classes for Java application on z/OS attempts to connect using client mode, and is not allowed to do so, MQRC_ENVIRONMENT_ERROR is returned.

Advanced Message Security (AMS) support

IBM MQ classes for Java client applications can use AMS when connecting to remote z/OS queue managers, subject to the conditions previously described in this topic.

To use AMS in this way, the client applications must use a key store type of `jceracfks` in `keystore.conf`, where:

- The property name prefix is `jceracfks` and this name prefix is case insensitive.
- The key store is a RACF keyring.
- Passwords are not required, and will be ignored. This is because RACF keyrings do not use passwords.
- If you specify the provider, the provider must be `IBMJCE`.

When you use `jceracfks` with AMS, the key store must be in the form: `safkeyring://`*`user`*`/`*`keyring`*, where:

- `safkeyring` is a literal and this name is case insensitive
- *`user`* is the RACF user id that owns the keyring
- *`keyring`* is the name of the RACF keyring and the name of the keyring is case sensitive

The following example uses the standard AMS keyring for user `JOHNDOE`:

```
jceracfks.keystore=safkeyring://JOHNDOE/drq.ams.keyring
```

Related concepts

"JMS/Jakarta Messaging client connectivity to batch applications running on z/OS" on page 121

Under certain conditions, an IBM MQ classes for JMS/Jakarta Messaging application on z/OS can connect to a queue manager on z/OS by using a client connection. Use of a client connection can simplify IBM MQ topologies.

Writing IBM MQ classes for Java applications

This collection of topics provides information to assist with writing Java applications to interact with IBM MQ systems.

To use IBM MQ classes for Java to access IBM MQ queues, you write Java applications that contain calls that put messages onto, and get messages from, IBM MQ queues. For details of individual classes, see IBM MQ classes for Java.

Note: Automatic client reconnection is not supported by IBM MQ classes for Java.

The IBM MQ classes for Java interface

The procedural IBM MQ application programming interface uses verbs, which act on objects. The Java programming interface uses objects, which you act upon by calling methods.

The procedural IBM MQ application programming interface is built around verbs such as these:

```
MQBACK, MQBEGIN, MQCLOSE, MQCONN, MQDISC,
MQGET, MQINQ, MQOPEN, MQPUT, MQSET, MQSUB
```

These verbs all take, as a parameter, a handle to the IBM MQ object on which they are to operate. Your program consists of a set of IBM MQ objects, which you act upon by calling methods on those objects.

When you use the procedural interface, you disconnect from a queue manager by using the call MQDISC(Hconn, CompCode, Reason), where *Hconn* is a handle to the queue manager.

In the Java interface, the queue manager is represented by an object of class MQQueueManager. You disconnect from the queue manager by calling the disconnect() method on that class.

```
// declare an object of type queue manager
MQQueueManager queueManager=new MQQueueManager();
...
// do something...
...
// disconnect from the queue manager
queueManager.disconnect();
```

IBM MQ classes for Java connection modes

The way you program for IBM MQ classes for Java has some dependencies on the connection modes you want to use.

If you use client connections, there are a number of differences from the IBM MQ MQI client but it is conceptually similar. If you use bindings mode you can use fastpath bindings and can issue the MQBEGIN command. You specify which mode to use by setting variables in the MQEnvironment class.

IBM MQ classes for Java client connections

When IBM MQ classes for Java is used as a client, it is like the IBM MQ MQI client, but has a number of differences.

If you are programming for *IBM MQ classes for Java* for use as a client, be aware of the following differences:

- It supports only TCP/IP.
- It does not read any IBM MQ environment variables at startup.
- Information that would be stored in a channel definition and in environment variables can be stored in a class called Environment. Alternatively, this information can be passed as parameters when the connection is made.

- Error and exception conditions are written to a log specified in the `MQException` class. The default error destination is the Java console.
- Only the following attributes in an IBM MQ client configuration file are relevant to IBM MQ classes for Java. If you specify other attributes, they are ineffective.

Stanza	Attribute
ClientExitPath stanza of the client configuration file	ExitsDefaultPath
ClientExitPath stanza of the client configuration file	ExitsDefaultPath64
ClientExitPath stanza of the client configuration file	JavaExitsClasspath
MessageBuffer stanza of the client configuration file	MaximumSize
MessageBuffer stanza of the client configuration file	PurgeTime
MessageBuffer stanza of the client configuration file	UpdatePercentage
TCP stanza of the client configuration file	ClntRcvBuffSize
TCP stanza of the client configuration file	ClntSndBuffSize
TCP stanza of the client configuration file	Connect_Timeout
TCP stanza of the client configuration file	KeepAlive

- If connecting to a queue manager that requires character data to be converted, then the V7 Java client is now capable of doing the conversion if queue manager is unable to do so. The client JVM must support the conversion between the CCSID of the client and that of the queue manager.
- Automatic client reconnect is not supported by IBM MQ classes for Java.

When used in client mode, *IBM MQ classes for Java* does not support the MQBEGIN call.

IBM MQ classes for Java bindings mode
The bindings mode of IBM MQ classes for Java differs from the client mode in three main ways.

When used in bindings mode, IBM MQ classes for Java uses the Java Native Interface (JNI) to call directly into the existing queue manager API, rather than communicating through a network.

By default, applications that use the IBM MQ classes for Java in bindings mode connect to a queue manager using the *ConnectOption*, MQCNO_STANDARD_BINDINGS.

The IBM MQ classes for Java support the following *ConnectOptions*:

- MQCNO_FASTPATH_BINDING
- MQCNO_STANDARD_BINDING
- MQCNO_SHARED_BINDING
- MQCNO_ISOLATED_BINDING

For further information on *ConnectOptions*, see "Connecting to a queue manager using the MQCONNX call" on page 712.

Bindings mode supports the MQBEGIN call to initiate global units of work that are coordinated by the queue manager, on all platforms apart from IBM MQ for IBM i and IBM MQ for z/OS.

Most of the parameters provided by the `MQEnvironment` class are not relevant to bindings mode and are ignored.

Defining which IBM MQ classes for Java connection to use
The type of connection to use is determined by the setting of variables in the MQEnvironment class.

Two variables are used:

MQEnvironment.properties
The connection type is determined by the value associated with the key name CMQC.TRANSPORT_PROPERTY. Possible values are as follows:

CMQC.TRANSPORT_MQSERIES_BINDINGS
Connect in bindings mode

CMQC.TRANSPORT_MQSERIES_CLIENT
Connect in client mode

CMQC.TRANSPORT_MQSERIES
Connection mode is determined by the value of the *hostname* property

MQEnvironment.hostname
Set the value of this variable as follows:

- For client connections, set the value of this variable to the host name of the IBM MQ server to which you want to connect
- For bindings mode, do not set this variable, or set it to null

Operations on queue managers
This collection of topics describes how to connect to, and disconnect from, a queue manager using IBM MQ classes for Java.

Setting up the IBM MQ environment for IBM MQ classes for Java
For an application to connect to a queue manager in client mode, the application must specify the channel name, host name, and port number.

Note: The information in this topic is relevant only if your application connects to a queue manager in client mode. It is not relevant if it connects in bindings mode. See: "Connection modes for IBM MQ classes for JMS" on page 105

You can specify the channel name, host name, and port number in one of two ways: either as fields in the MQEnvironment class or as properties of the MQQueueManager object.

If you set fields in the MQEnvironment class, they apply to your whole application, except where they are overridden by a properties hash table. To specify the channel name and host name in MQEnvironment, use the following code:

```
MQEnvironment.hostname = "host.domain.com";
MQEnvironment.channel = "java.client.channel";
```

This is equivalent to setting an **MQSERVER** environment variable:

```
"java.client.channel/TCP/host.domain.com".
```

By default, the Java clients attempt to connect to an IBM MQ listener at port 1414. To specify a different port, use the following code:

```
MQEnvironment.port = nnnn;
```

where nnnn is the required port number

If you pass properties to a queue manager object at its creation, they apply only to that queue manager. Create entries in a Hashtable object with keys of **hostname**, **channel**, and, optionally, **port**, and with appropriate values. To use the default port, 1414, you can omit the **port** entry. Create the MQQueueManager object by using a constructor that accepts the properties hash table.

Identifying a connection to the queue manager by setting an application name

An application can set a name that identifies its connection to the queue manager. This application name is shown by the **DISPLAY CONN MQSC/PCF** command (where the field is called **APPLTAG**) or in the IBM MQ Explorer **Application Connections** display (where the field is called **App name**).

Application names are limited to 28 characters, so longer names are truncated. If an application name is not specified, a default is provided. The default name is based on the invoking (main) class, but if this information is not available, the text `IBM MQ Client for Java` is used.

If the name of the invoking class is used, it is adjusted to fit by removing leading package names, if necessary. For example, if the invoking class is `com.example.MainApp`, the full name is used, but if the invoking class is `com.example.dictionaryAndThesaurus.multilingual.mainApp`, the name `multilingual.mainApp` is used, because it is the longest combination of class name and rightmost package name that fits into the available length.

If the class name itself is more than 28 characters long, it is truncated to fit. For example, `com.example.mainApplicationForSecondTestCase` becomes `mainApplicationForSecondTest`.

To set an application name in the MQEnvironment class, add the name to the MQEnvironment.properties hash table, with a key of **MQConstants.APPNAME_PROPERTY**, by using the following code:

```
MQEnvironment.properties.put(MQConstants.APPNAME_PROPERTY, "my_application_name");
```

To set an application name in the properties hash table that is passed to the MQQueueManager constructor, add the name to the properties hash table with a key of **MQConstants.APPNAME_PROPERTY**.

Overriding properties specified in an IBM MQ client configuration file

An IBM MQ client configuration file can also specify properties that are used to configure IBM MQ classes for Java. However, properties specified in an IBM MQ MQI client configuration file apply only when an application connects to a queue manager in client mode.

If required, you can override any attribute in an IBM MQ configuration file in any of the following ways. The options are shown in order of precedence.

- Set a Java system property for the configuration property.
- Set the property in the MQEnvironment.properties map.
- On Java5 and later releases, set a system environment variable.

Only the following attributes in an IBM MQ client configuration file are relevant to IBM MQ classes for Java. If you specify or override other attributes, it has no effect.

Stanza	Attribute
ClientExitPath stanza of the client configuration file	ExitsDefaultPath
ClientExitPath stanza of the client configuration file	ExitsDefaultPath64
ClientExitPath stanza of the client configuration file	JavaExitsClasspath
MessageBuffer stanza of the client configuration file	MaximumSize
MessageBuffer stanza of the client configuration file	PurgeTime
MessageBuffer stanza of the client configuration file	UpdatePercentage
TCP stanza of the client configuration file	ClntRcvBufSize

Stanza	Attribute
TCP stanza of the client configuration file	ClntSndBufSize
TCP stanza of the client configuration file	Connect_Timeout
TCP stanza of the client configuration file	KeepAlive

Connecting to a queue manager in IBM MQ classes for Java
Connect to a queue manager by creating a new instance of the MQQueueManager class. Disconnect from a queue manager by calling the disconnect() method.

You are now ready to connect to a queue manager by creating a new instance of the MQQueueManager class:

```
MQQueueManager queueManager = new MQQueueManager("qMgrName");
```

To disconnect from a queue manager, call the disconnect() method on the queue manager:

```
queueManager.disconnect();
```

If you call the disconnect method, all open queues and processes that you have accessed through that queue manager are closed. However, it is good programming practice to close these resources explicitly when you finish using them. To do this, use the close() method on the relevant objects.

The commit() and backout() methods on a queue manager are equivalent to the MQCMIT and MQBACK calls that are used with the procedural interface.

Using a client channel definition table with IBM MQ classes for Java
An IBM MQ classes for Java client application can use client connection channel definitions stored in a client channel definition table (CCDT).

As an alternative to creating a client connection channel definition by setting certain fields and environment properties in the `MQEnvironment` class or passing them to an `MQQueueManager` in a properties hash table, an IBM MQ classes for Java client application can use client connection channel definitions that are stored in a client channel definition table. These definitions are created by IBM MQ Script (MQSC) commands or IBM MQ Programmable Command Format (PCF) commands, or using the IBM MQ Explorer .

When the application creates an `MQQueueManager` object, the IBM MQ classes for Java client searches the client channel definition table for a suitable client connection channel definition, and uses the channel definition to start an MQI channel. For more information about client channel definition tables and how to construct one, see Client channel definition table.

To use a client channel definition table, an application must first create a URL object. The URL object encapsulates a uniform resource locator (URL) that identifies the name and location of the file containing the client channel definition table and specifies how the file can be accessed.

For example, if the file `ccdt1.tab` contains a client channel definition table and is stored on the same system on which the application is running, the application can create a URL object in the following way:

```
java.net.URL chanTab1 = new URL("file:///home/admdata/ccdt1.tab");
```

As another example, suppose the file `ccdt2.tab` contains a client channel definition table and is stored on a system that is different from the one on which the application is running. If the file can be accessed using the FTP protocol, the application can create a URL object in the following way:

```
java.net.URL chanTab2 = new URL("ftp://ftp.server/admdata/ccdt2.tab");
```

After the application has created a URL object, the application can create an `MQQueueManager` object using one of the constructors that takes a URL object as a parameter. Here is an example:

```
MQQueueManager mars = new MQQueueManager("MARS", chanTab2);
```

This statement causes the IBM MQ classes for Java client to access the client channel definition table identified by the URL object chanTab2, search the table for a suitable client connection channel definition, and then use the channel definition to start an MQI channel to the queue manager called MARS.

Note the following points that apply if an application uses a client channel definition table:

- When the application creates an `MQQueueManager` object using a constructor that takes a URL object as a parameter, no channel name must be set in the `MQEnvironment` class, either as a field or as an environment property. If a channel name is set, the IBM MQ classes for Java client throws an `MQException`. The field or environment property specifying the channel name is considered to be set if its value is anything other than null, an empty string, or a string containing all blank characters.
- The **queueManagerName** parameter on the `MQQueueManager` constructor can have one of the following values:
  - The name of a queue manager
  - An asterisk (*) followed by the name of a queue manager group
  - An asterisk (*)
  - Null, an empty string, or a string containing all blank characters

  These are the same values that can be used for the **QMgrName** parameter on an MQCONN call issued by a client application that is using Message Queue Interface (MQI). For more information about the meaning of these values, see"The Message Queue Interface overview" on page 697.

  If your application uses connection pooling, see"Controlling the default connection pool in IBM MQ classes for Java" on page 383.
- When the IBM MQ classes for Java client finds a suitable client connection channel definition in the client channel definition table, it uses only the information extracted from this channel definition to start an MQI channel. Any channel related fields or environment properties that the application might have set in the `MQEnvironment` class are ignored.

  In particular, note the following points if you are using Transport Layer Security (TLS):
  - An MQI channel uses TLS only if the channel definition extracted from the client channel definition table specifies the name of a CipherSpec supported by the IBM MQ classes for Java client.
  - A client channel definition table also contains information about the location of Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs). The IBM MQ classes for Java client uses only this information to access LDAP servers that hold CRLs.
  - A client channel definition table can also contain the location of an OCSP responder. IBM MQ classes for Java cannot use the OCSP information in a client channel definition table file. However, you can configure OCSP as described in the section Using Online Certificate Protocol

  For more information about using TLS with a client channel definition table, see Specifying that an MQI channel uses TLS.

  Note also the following points if you are using channel exits:
  - An MQI channel uses the channel exits and associated user data specified by the channel definition extracted from the client channel definition table in preference to channel exits and data specified using other methods.
  - A channel definition extracted from a client channel definition table can specify channel exits that are written in Java, C, or C++. For more information about how to write a channel exit in Java , see"Creating a channel exit in IBM MQ classes for Java" on page 377. For more information about how to write a channel exit in other languages, see "Using channel exits not written in Java with IBM MQ classes for Java" on page 380.

Specifying a range of ports for IBM MQ classes for Java client connections
You can specify a port, or a range of ports, that an application can bind to in either of two ways.

When an IBM MQ classes for Java application attempts to connect to an IBM MQ queue manager in client mode, a firewall might allow only those connections that originate from specified ports or range of ports. In this situation, you can specify a port, or a range of ports, that the application can bind to. You can specify the port(s) in the following ways:

- You can set the localAddressSetting field in the MQEnvironment class. Here is an example:

```
MQEnvironment.localAddressSetting = "192.0.2.0(2000,3000)";
```

- You can set the environment property CMQC.LOCAL_ADDRESS_PROPERTY. Here is an example:

```
(MQEnvironment.properties).put(CMQC.LOCAL_ADDRESS_PROPERTY,
                               "192.0.2.0(2000,3000)");
```

- When you can construct the MQQueueManager object, you can pass a properties hashtable containing a LOCAL_ADDRESS_PROPERTY with the value "192.0.2.0(2000,3000)"

In each of these examples, when the application later connects to a queue manager, the application binds to a local IP address and port number in the range 192.0.2.0(2000) to 192.0.2.0(3000).

In a system with more than one network interface, you can also use the localAddressSetting field, or the environment property CMQC.LOCAL_ADDRESS_PROPERTY, to specify which network interface must be used for a connection.

Connection errors might occur if you restrict the range of ports. If an error occurs, an MQException is thrown containing the IBM MQ reason code MQRC_Q_MGR_NOT_AVAILABLE and the following message:

```
Socket connection attempt refused due to LOCAL_ADDRESS_PROPERTY restrictions
```

An error might occur if all the ports in the specified range are in use, or if the specified IP address, host name, or port number is not valid (a negative port number, for example).

Accessing queues, topics, and processes in IBM MQ classes for Java

To access queues, topics, and processes, use methods of the MQQueueManager class. The MQOD (object descriptor structure) is collapsed into the parameters of these methods.

Queues

To open a queue you can use the accessQueue method of the MQQueueManager class. For example, on a queue manager called queueManager, use the following code:

```
MQQueue queue = queueManager.accessQueue("qName",CMQC.MQOO_OUTPUT);
```

The accessQueue method returns a new object of class MQQueue.

When you have finished using the queue, use the close() method to close it, as in the following example:

```
queue.close();
```

You can also create a queue by using the MQQueue constructor. The parameters are exactly the same as for the accessQueue method, with the addition of a queue manager parameter. For example:

```
MQQueue queue = new MQQueue(queueManager,
                            "qName",
                            CMQC.MQOO_OUTPUT,
                            "qMgrName",
```

```
                                "dynamicQName",
                                "altUserID");
```

You can specify a number of options when you create queues. For details of these, see Class.com.ibm.mq.MQQueue. Constructing a queue object in this way enables you to write your own subclasses of MQQueue.

Topics

Similarly, you can open a topic using the accessTopic method of the MQQueueManager class. For example, on a queue manager called queueManager, use the following code to create a subscriber and publisher:

```
MQTopic subscriber =
      queueManager.accessTopic("TOPICSTRING","TOPICNAME",
      CMQC.MQTOPIC_OPEN_AS_SUBSCRIPTION, CMQC.MQSO_CREATE);
```

```
MQTopic publisher =
      queueManager.accessTopic("TOPICSTRING","TOPICNAME",
      CMQC.MQTOPIC_OPEN_AS_PUBLICATION, CMQC.MQOO_OUTPUT);
```

When you have finished using the topic, use the close() method to close it.

You can also create a topic by using the MQTopic constructor. The parameters are exactly the same as for the accessTopic method, with the addition of a queue manager parameter. For example:

```
MQTopic subscriber = new
      MQTopic(queueManager,"TOPICSTRING","TOPICNAME",
      CMQC.MQTOPIC_OPEN_AS_SUBSCRIPTION, CMQC.MQSO_CREATE);
```

You can specify a number of options when you create topics. For details of these, see Class com.ibm.mq.MQTopic. Constructing a topic object in this way enables you to write your own subclasses of MQTopic.

A topic must be opened either for publication or for subscription. The MQQueueManager class has eight accessTopic methods and the Topic class has eight constructors. In each case, four of these have a **destination** parameter and four have a **subscriptionName** parameter (including two that have both). These can only be used to open the topic for subscriptions. The two remaining methods have an **openAs** parameter, and the topic can be opened for either publication or subscription depending on the value of the **openAs** parameter.

To create a topic as a durable subscriber use either an accessTopic method of the MQQueueManager class or an MQTopic constructor that accepts a subscription name and, in either case, set the CMQC.MQSO_DURABLE option.

Processes

To access a process, use the accessProcess method of the MQQueueManager. For example, on a queue manager called queueManager, use the following code to create an MQProcess object:

```
MQProcess process =
queueManager.accessProcess("PROCESSNAME",
CMQC.MQOO_FAIL_IF_QUIESCING);
```

To access a process, use the accessProcess method of the MQQueueManager.

The accessProcess method returns a new object of class MQProcess.

When you have finished using the process object, use the close() method to close it, as in the following example:

```
process.close();
```

You can also create a process by using the MQProcess constructor. The parameters are exactly the same as for the accessProcess method, with the addition of a queue manager parameter. For example:

```
MQProcess process =
      new MQProcess(queueManager,"PROCESSNAME",
      CMQC.MQOO_FAIL_IF_QUIESCING);
```

Constructing a process object in this way enables you to write your own subclasses of MQProcess.

Handling messages in IBM MQ classes for Java

Messages are represented by the MQMessage class. You put and get messages using methods of the MQDestination class, which has subclasses of MQQueue and MQTopic.

Put messages onto queues or topics using the put() method of the MQDestination class. You get messages from queues or topics using the get() method of the MQDestination class. Unlike the procedural interface, where MQPUT and MQGET put and get arrays of bytes, the Java programming language puts and gets instances of the MQMessage class. The MQMessage class encapsulates the data buffer that contains the actual message data, together with all the MQMD (message descriptor) parameters and message properties that describe that message.

To build a new message, create a new instance of the MQMessage class, and use the writeXXX methods to put data into the message buffer.

When the new message instance is created, all the MQMD parameters are automatically set to their default values, as defined in Initial values and language declarations for MQMD. The put() method of MQDestination also takes an instance of the MQPutMessageOptions class as a parameter. This class represents the MQPMO structure. The following example creates a message and puts it onto a queue:

```
// Build a new message containing my age followed by my name
MQMessage myMessage = new MQMessage();
myMessage.writeInt(25);

String name = "Charlie Jordan";
myMessage.writeInt(name.length());
myMessage.writeBytes(name);

// Use the default put message options...
MQPutMessageOptions pmo = new MQPutMessageOptions();

// put the message
!queue.put(myMessage,pmo);
```

The get() method of MQDestination returns a new instance of MQMessage, which represents the message just taken from the queue. It also takes an instance of the MQGetMessageOptions class as a parameter. This class represents the MQGMO structure.

You do not need to specify a maximum message size, because the get() method automatically adjusts the size of its internal buffer to fit the incoming message. Use the readXXX methods of the MQMessage class to access the data in the returned message.

The following example shows how to get a message from a queue:

```
// Get a message from the queue
MQMessage theMessage    = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();
queue.get(theMessage,gmo); // has default values

// Extract the message data
int age = theMessage.readInt();
int strLen = theMessage.readInt();
byte[] strData = new byte[strLen];
theMessage.readFully(strData,0,strLen);
String name = new String(strData,0);
```

You can alter the number format that the read and write methods use by setting the *encoding* member variable.

You can alter the character set to use for reading and writing strings by setting the *characterSet* member variable.

See MQMessage class for more information.

Note: The writeUTF() method of MQMessage automatically encodes the length of the string as well as the Unicode bytes it contains. When your message will be read by another Java program (using readUTF()), this is the simplest way to send string information.

Improving the performance of nonpersistent messages in IBM MQ classes for Java
To improve performance when browsing messages or consuming nonpersistent messages from a client application, you can use *read ahead*. Client applications using MQGET or asynchronous consumption will benefit from the performance improvements when browsing messages or consuming nonpersistent messages.

For general information about the read ahead facility, see the related topic.

In IBM MQ classes for Java, you use the CMQC.MQSO_READ_AHEAD and CMQC.MQSO_NO_READ_AHEAD properties of an MQQueue or MQTopic object to determine whether message consumers and queue browsers are allowed to use read ahead on that object.

Putting messages asynchronously using IBM MQ classes for Java
To put a message asynchronously, set MQPMO_ASYNC_RESPONSE.

You put messages onto queues or topics using the put() method of the MQDestination class.
To put a message asynchronously, that is, allowing the operation to complete without waiting for a response from the queue manager, you can set MQPMO_ASYNC_RESPONSE in the options field of MQPutMessageOptions. To determine the success or failure of asynchronous puts, use the MQQueueManager.getAsyncStatus call.

Publish/subscribe in IBM MQ classes for Java
In IBM MQ classes for Java, the topic is represented by the MQTopic class, and you publish to it using the MQTopic.put() methods.

For general information about IBM MQ publish/subscribe, see Publish/subscribe messaging.

Handling IBM MQ message headers with IBM MQ classes for Java
Java classes are provided representing different types of message header. Two helper classes are also provided.

The MQHeader interface

Header objects are described by the MQHeader interface, which provides general-purpose methods for accessing header fields and for reading and writing message content. Each header type has its own class that implements the MQHeader interface and adds getter and setter methods for individual fields. For example, the MQRFH2 header type is represented by the MQRFH2 class; the MQDLH header type by the MQDLH class, and so on. The header classes perform any necessary data conversion automatically, and can read or write data in any specified numeric encoding or character set (CCSID).

Important: The MQRFH2 headers classes treat the message as a random access file, which means that the cursor must be positioned at the start of the message. Before using an internal message header class like MQRFH, MQRFH2, MQCIH, MQDEAD, MQIIH or MQXMIT, make sure that you update the message's cursor position to the correct location before passing the message to the class.

Helper classes

Two helper classes, MQHeaderIterator and MQHeaderList, assist with reading and decoding (parsing) the header content in messages:

- The MQHeaderIterator class works like a java.util.Iterator. For as long as there are more headers in the message, the next() method returns true, and the nextHeader() or next() method returns the next header object.
- The MQHeaderList works like a java.util.List. Like the MQHeaderIterator, it parses header content, but it also allows you to search for particular headers, add new headers, remove existing headers, update header fields and then write the header content back to a message. Alternatively, you can create an empty MQHeaderList, then populate it with header instances and write it to a message once or repeatedly.

The MQHeaderIterator and MQHeaderList classes use the information in the MQHeaderRegistry to know which IBM MQ header classes are associated with particular message types and formats. The MQHeaderRegistry is configured with knowledge of all current IBM MQ formats and header types and their implementation classes, and you can also register your own header types.

Support is provided for the following commonly used IBM MQ headers

- MQRFH - Rules and formatting header
- MQRFH2 - Like MQRFH, used to pass messages to and from a message broker belonging to IBM Integration Bus. Also used to contain message properties
- MQCIH - CICS Bridge
- MQDLH - Dead letter header
- MQIIH - IMS information header
- MQRMH - reference message header
- MQSAPH - SAP header
- MQWIH - Work information header
- MQXQH - Transmission Queue header
- MQDH - Distribution header
- MQEPH - Encapsulated PCF header

You can also define classes representing your own headers.

To use an MQHeaderIterator to get an RFH2 header, either set MQGMO_PROPERTIES_FORCE_MQRFH2 in the GetMessageOptions, or set the queue property PROPCTL to FORCE.

Printing all the headers in a message using IBM MQ classes for Java
In this example, an instance of MQHeaderIterator parses the headers in an MQMessage that has been received from a queue. The MQHeader objects returned from the nextHeader() method display their structure and contents when their toString method is invoked.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeader;
import com.ibm.mq.headers.MQHeaderIterator;
...
MQMessage message = ... // Message received from a queue.
MQHeaderIterator it = new MQHeaderIterator (message);

while (it.hasNext ())
{
    MQHeader header = it.nextHeader ();

    System.out.println ("Header type " + header.type () + ": " + header);
}
```

Skipping over the headers in a message using IBM MQ classes for Java
In this example, the skipHeaders() method of MQHeaderIterator positions the message read cursor immediately after the last header.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeaderIterator;
...
MQMessage message = ... // Message received from a queue.
MQHeaderIterator it = new MQHeaderIterator (message);

it.skipHeaders ();
```

Finding the reason code in a dead-letter message using IBM MQ classes for Java
In this example, the read method populates the MQDLH object by reading from the message. After the read operation, the message read cursor is positioned immediately after the MQDLH header content.

Messages on the queue manager's dead-letter queue are prefixed with a dead-letter header (MQDLH). To decide how to handle these messages - for example, to determine whether to retry or discard them - a dead-letter handling application must look at the reason code contained in the MQDLH.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQDLH;
...
MQMessage message = ... // Message received from the dead-letter queue.
MQDLH dlh = new MQDLH ();

dlh.read (message);

System.out.println ("Reason: " + dlh.getReason ());
```

All header classes also provide a convenience constructor to initialize themselves directly from the message in a single step. So the code in this example could be simplified as follows:

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQDLH;
...
MQMessage message = ... // Message received from the dead-letter queue.
MQDLH dlh = new MQDLH (message);

System.out.println ("Reason: " + dlh.getReason ());
```

Reading and removing the header from a dead-letter message using IBM MQ classes for Java
In this example, MQDLH is used to remove the header from a dead-letter message.

A dead-letter handling application will typically resubmit messages that have been rejected if their reason code indicates a transient error. Before resubmitting the message, it must remove the MQDLH header.

This example performs the following steps (see the comments in the example code):

1. The MQHeaderList reads the entire message, and each header encountered in the message becomes an item in the list.
2. Dead-letter messages contain an MQDLH as their first header, so this can be found in the first item of the header list. The MQDLH has already been populated from the message when the MQHeaderList is built, so there is no need to invoke its read method.
3. The reason code is extracted using the getReason() method provided by the MQDLH class.
4. The reason code has been inspected, and indicates that it is appropriate to resubmit the message. The MQDLH is removed using the MQHeaderList remove() method.
5. The MQHeaderList writes its remaining content to a new message object. The new message now contains everything in the original message except the MQDLH and can be written to a queue. The **true** argument to the constructor and to the write method indicates that the message body is to be held within the MQHeaderList, and written out again.

6. The format field in the message descriptor of the new message now contains the value that was previously in the MQDLH format field. The message data matches the numeric encoding and CCSID set in the message descriptor.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQDLH;
import com.ibm.mq.headers.MQHeaderList;
...
MQMessage message = ... // Message received from the dead-letter queue.
MQHeaderList list = new MQHeaderList (message, true); // Step 1.
MQDLH dlh = (MQDLH) list.get (0); // Step 2.
int reason = dlh.getReason (); // Step 3.
...
list.remove (dlh); // Step 4.

MQMessage newMessage = new MQMessage ();

list.write (newMessage, true); // Step 5.
newMessage.format = list.getFormat (); // Step 6.
```

Printing the content of a message using IBM MQ classes for Java
This example uses MQHeaderList to print out the content of a message, including its headers.

The output contains a view of all the header contents as well as the body of the message. The MQHeaderList class decodes all the headers in one go, whereas the MQHeaderIterator steps through them one at a time under application control. You might use this technique to provide a simple debugging tool when writing WebSphere MQ applications.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeaderList;
...
MQMessage message = ... // Message received from a queue.

System.out.println (new MQHeaderList (message, true));
```

This example also prints out the message descriptor fields, using the MQMD class. The copyFrom() method of the com.ibm.mq.headers.MQMD class populates the header object from the message descriptor fields of the MQMessage rather than by reading the message body.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQMD;
import com.ibm.mq.headers.MQHeaderList;
...
MQMessage message = ...
MQMD md = new MQMD ();
...
md.copyFrom (message);
System.out.println (md + "\n" + new MQHeaderList (message, true));
```

Finding a specific type of header in a message using IBM MQ classes for Java
This example uses the indexOf(String) method of MQHeaderList to find an MQRFH2 header in a message, if one is present.

```
import com.ibm.mq.MQMessage;
import com.ibm.mq.headers.MQHeaderList;
import com.ibm.mq.headers.MQRFH2;
...
MQMessage message = ...
MQHeaderList list = new MQHeaderList (message);
int index = list.indexOf ("MQRFH2");

if (index >= 0)
{
    MQRFH2 rfh = (MQRFH2) list.get (index);
        ...
}
```

Analyzing an MQRFH2 header using IBM MQ classes for Java
This example shows how to access a known field value in a named folder, using the MQRFH2 class.

The MQRFH2 class provides a number of ways to access not only the fields in the fixed part of the structure, but also the XML-encoded folder contents that are carried within the NameValueData field. This example shows how to access a known field value in a named folder - in this instance, the Rto field in the jms folder, which represents the reply queue name in an MQ JMS message.

```
MQRFH2 rfh = ...

String value = rfh.getStringFieldValue ("jms", "Rto");
```

To discover the contents of an MQRFH2 (as opposed to requesting specific fields directly), you can use the getFolders method to return a list of MQRFH2.Element, which represents the structure of a folder that could contain fields and other folders. Setting a field or folder to null removes it from the MQRFH2. When you manipulate the NameValueData folder contents in this way, the StrucLength field is automatically updated accordingly.

Reading and writing byte streams other than MQMessage objects using IBM MQ classes for Java
These examples use the header classes to parse and manipulate IBM MQ header content when the data source is not an MQMessage object.

You can use the header classes to parse and manipulate IBM MQ header content even when the data source is something other than an MQMessage object. The MQHeader interface implemented by every header class provides the methods `int read (java.io.DataInput message, int encoding, int characterSet)` and `int write (java.io.DataOutput message, int encoding, int characterSet)`. The com.ibm.mq.MQMessage class implements the java.io.DataInput and java.io.DataOutput interfaces. This means that you can use the two MQHeader methods to read and write MQMessage content, overriding the encoding and CCSID specified in the message descriptor. This is useful for messages that contain a chain of headers in different encodings.

You can also obtain DataInput and DataOutput objects from other data streams, for example file or socket streams, or byte arrays carried in JMS messages. The java.io.DataInputStream classes implement DataInput and the java.io.DataOutputStream classes implement DataOutput. This example reads IBM MQ header content from a byte array:

```
import java.io.*;
import com.ibm.mq.headers.*;
...
byte [] bytes = ...
DataInput in = new DataInputStream (new ByteArrayInputStream (bytes));
MQHeaderIterator it = new MQHeaderIterator (in, CMQC.MQENC_NATIVE,
  CMQC.MQCCSI_DEFAULT);
```

The line starting MQHeaderIterator could be replaced with

```
MQDLH dlh = new MQDLH (in, CMQC.MQENC_NATIVE, CMQC.MQCCSI_DEFAULT);
// or any other header type
```

This example writes to a byte array using a DataOutputStream:

```
MQHeader header = ... // Could be any header type
ByteArrayOutputStream out = new ByteArrayOutputStream ();

header.write (new DataOutputStream (out), CMQC.MQENC_NATIVE, CMQC.MQCCSI_DEFAULT);
byte [] bytes = out.toByteArray ();
```

When you work with streams in this way, be careful to use the correct values for the encoding and characterSet arguments. When reading headers, specify the encoding and CCSID with which the byte content was originally written. When writing headers, specify the encoding and CCSID that you want to produce. The data conversion is performed automatically by the header classes.

Creating classes for new header types using IBM MQ classes for Java
You can create Java classes for header types not supplied with IBM MQ classes for Java.

To add a Java class representing a new header type that you can use in the same way as any header class supplied with IBM MQ classes for Java, you create a class that implements the MQHeader interface. The simplest approach is to extend the com.ibm.mq.headers.impl.Header class. This example produces a fully-functional class representing the MQTM header structure. You do not have to add individual getter and setter methods for each field, but it is a useful convenience for users of the header class. The generic getValue and setValue methods that take a string for the field name will work for all fields defined in the header type. The inherited read, write and size methods will enable instances of the new header type to be read and written and will calculate the header size correctly based upon its field definition. The type definition is created just once, however many instances of this header class are created. To make the new header definition available for decoding using the MQHeaderIterator or MQHeaderList classes, you would register it using the MQHeaderRegistry. Note however that the MQTM header class is in fact already provided in this package and registered in the default registry.

```
import com.ibm.mq.headers.impl.Header;
import com.ibm.mq.headers.impl.HeaderField;
import com.ibm.mq.headers.CMQC;

public class MQTM extends Header {
    final static HeaderType TYPE = new HeaderType ("MQTM");
    final static HeaderField StrucId = TYPE.addMQChar ("StrucId", CMQC.MQTM_STRUC_ID);
    final static HeaderField Version = TYPE.addMQLong ("Version", CMQC.MQTM_VERSION_1);
    final static HeaderField QName = TYPE.addMQChar ("QName", CMQC.MQ_Q_NAME_LENGTH);
    final static HeaderField ProcessName = TYPE.addMQChar ("ProcessName",
      CMQC.MQ_PROCESS_NAME_LENGTH);
    final static HeaderField TriggerData = TYPE.addMQChar ("TriggerData",
      CMQC.MQ_TRIGGER_DATA_LENGTH);
    final static HeaderField ApplType = TYPE.addMQLong ("ApplType");
    final static HeaderField ApplId = TYPE.addMQChar ("ApplId", 256);
    final static HeaderField EnvData = TYPE.addMQChar ("EnvData", 128);
    final static HeaderField UserData = TYPE.addMQChar ("UserData", 128);

    protected MQTM (HeaderType type){
        super (type);
    }
    public String getStrucId () {
        return getStringValue (StrucId);
    }
    public int getVersion () {
        return getIntValue (Version);
    }
    public String getQName () {
        return getStringValue (QName);
    }
    public void setQName (String value) {
        setStringValue (QName, value);
    }
    // ...Add convenience getters and setters for remaining fields in the same way.
}
```

Handling PCF messages with IBM MQ classes for Java
Java classes are provided to create and parse PCF-structured messages, and to facilitate sending PCF requests and collecting PCF responses.

Classes PCFMessage & MQCFGR represent arrays of PCF parameter structures. They provide convenience methods for adding and retrieving PCF parameters.

PCF parameter structures are represented by the classes MQCFH, MQCFIN, MQCFIN64, MQCFST, MQCFBS, MQCFIL, MQCFIL64 MQCFSL, and MQCFGR. These share basic operational interfaces:

- Methods to read and write message content: read (), write (), and size ()
- Methods to manipulate parameters: getValue (), setValue (), getParameter () and others
- The enumerator method .nextParameter (), which parses PCF content in an MQMessage

The PCF filter parameter is used in inquire commands to provide a filter function. It in encapsulated in the following classes:

- MQCFIF - integer filter
- MQCFSF - string filter
- MQCFBF - byte filter

Two agent classes, PCFAgent and PCFMessageAgent are provided to manage the connection to a Queue Manager, the command server queue, and an associated response queue. PCFMessageAgent extends PCFAgent and should normally be used in preference to it. The PCFMessageAgent class converts the received MQMessages and passes them back to the caller as a PCFMessage array. PCFAgent returns an array of MQMessages, which you have to parse before use.

Handling message properties in IBM MQ classes for Java

Function calls to process message handles have no equivalent in IBM MQ classes for Java. To set, return, or delete message handle properties, use methods of the MQMessage class.

For general information about message properties, see “Property names” on page 27.

In IBM MQ classes for Java access to messages is through the MQMessage class. Message handles are therefore not provided in the Java environment and there is no equivalent to the IBM MQ function calls MQCRTMH, MQDLTMH, MQMHBUF, and MQBUFMH

To set message handle properties in the procedural interface, you use the call MQSETMP. In IBM MQ classes for Java, use the appropriate method of the MQMessage class:

- setBooleanProperty
- setByteProperty
- setBytesProperty
- setShortProperty
- setIntProperty
- setInt2Property
- setInt4Property
- setInt8Property
- setLongProperty
- setFloatProperty
- setDoubleProperty
- setStringProperty
- setObjectProperty

These are sometimes referred to collectively as the *set*property* methods.

To return the value of message handle properties in the procedural interface, you use the call MQINQMP. In IBM MQ classes for Java, use the appropriate method of the MQMessage class:

- getBooleanProperty
- getByteProperty
- getBytesProperty
- getShortProperty
- getIntProperty
- getInt2Property
- getInt4Property
- getInt8Property
- getLongProperty
- getFloatProperty
- getDoubleProperty

- getStringProperty
- getObjectProperty

These are sometimes referred to collectively as the *get*property* methods.

To delete the value of message handle properties in the procedural interface, you use the call MQDLTMP. In IBM MQ classes for Java, use the deleteProperty method of the MQMessage class.

Handling errors in IBM MQ classes for Java

Handle errors arising from IBM MQ classes for Java using Java `try` and `catch` blocks.

Methods in the Java interface do not return a completion code and reason code. Instead, they throw an exception whenever the completion code and reason code resulting from an IBM MQ call are not both zero. This simplifies the program logic so that you do not have to check the return codes after each call to IBM MQ. You can decide at which points in your program you want to deal with the possibility of failure. At these points, you can surround your code with `try` and `catch` blocks, as in the following example:

```
try {
    myQueue.put(messageA,putMessageOptionsA);
    myQueue.put(messageB,putMessageOptionsB);
}
catch (MQException ex) {
    // This block of code is only executed if one of
    // the two put methods gave rise to a non-zero
    // completion code or reason code.
    System.out.println("An error occurred during the put operation:" +
                       "CC = " + ex.completionCode +
                       "RC = " + ex.reasonCode);
    System.out.println("Cause exception:" + ex.getCause() );
}
```

The IBM MQ call reason codes reported back in Java exceptions for z/OS are documented in API completion and reason codes.

Exceptions that are thrown while an IBM MQ classes for Java application is running are also written to the log. However, an application can call the MQException.logExclude() method to prevent exceptions associated with a specific reason code from being logged. You might want to do this in situations where you expect many exceptions associated with a specific reason code to be thrown, and you do not want the log to be filled with these exceptions. For example, if your application attempts to get a message from a queue each time it iterates around a loop and, for most of these attempts, you expect no suitable message to be on the queue, you might want to prevent exceptions associated with the reason code MQRC_NO_MSG_AVAILABLE from being logged. If an application has previously prevented exceptions associated with a specific reason code from being logged, it can allow these exceptions to be logged again by calling the method MQException.logInclude().

Sometimes the reason code does not convey all details associated with the error. For each exception that is thrown, an application should check the linked exception. The linked exception itself might have another linked exception, and so the linked exceptions form a chain leading back to the original underlying problem. A linked exception is implemented by using the chained exception mechanism of the java.lang.Throwable class, and an application obtains a linked exception by calling the Throwable.getCause() method. From an exception that is an instance of MQException, MQException.getCause() retrieves the underlying instance of com.ibm.mq.jmqi.JmqiException, and getCause from this exception retrieves the underlying java.lang.Exception that caused the error.

Getting and setting attribute values in IBM MQ classes for Java

getXXX() and setXXX() methods are provided for many common attributes. Others can be accessed using the generic inquire() and set() methods.

For many of the common attributes, the classes MQManagedObject, MQDestination, MQQueue, MQTopic, MQProcess, and MQQueueManager contain getXXX() and setXXX() methods. These methods allow you to get and set their attribute values. Note that for MQDestination, MQQueue, and MQTopic, the methods work only if you specify the appropriate inquire and set flags when you open the object.

For less common attributes, the MQQueueManager, MQDestination, MQQueue, MQTopic,, and MQProcess classes all inherit from a class called MQManagedObject. This class defines the inquire() and set() interfaces.

When you create a new queue manager object by using the *new* operator, it is automatically opened for inquire. When you use the accessProcess() method to access a process object, that object is automatically opened for inquire. When you use the accessQueue() method to access a queue object, that object is not automatically opened for either inquire or set operations. This is because adding these options automatically can cause problems with some types of remote queues. To use the inquire, set, getXXX, and setXXX methods on a queue, you must specify the appropriate inquire and set flags in the openOptions parameter of the accessQueue() method. The same is true for destination and topic objects.

The inquire and set methods take three parameters:

- selectors array
- intAttrs array
- charAttrs array

You do not need the SelectorCount, IntAttrCount, and CharAttrLength parameters that are found in MQINQ, because the length of an array in Java is always known. The following example shows how to make an inquiry on a queue:

```
// inquire on a queue
final static int MQIA_DEF_PRIORITY = 6;
final static int MQCA_Q_DESC = 2013;
final static int MQ_Q_DESC_LENGTH = 64;

int[] selectors  = new int[2];
int[] intAttrs   = new int[1];
byte[] charAttrs = new byte[MQ_Q_DESC_LENGTH]

selectors[0] = MQIA_DEF_PRIORITY;
selectors[1] = MQCA_Q_DESC;

queue.inquire(selectors,intAttrs,charAttrs);

System.out.println("Default Priority = " + intAttrs[0]);
System.out.println("Description : " + new String(charAttrs,0));
```

Multithreaded programs in Java

The Java runtime environment is inherently multithreaded. IBM MQ classes for Java allows a queue manager object to be shared by multiple threads but ensures that all access to the target queue manager is synchronized.

Multithreaded programs are hard to avoid in Java. Consider a simple program that connects to a queue manager and opens a queue at startup. The program displays a single button on the screen. When a user clicks that button, the program fetches a message from the queue.

The Java runtime environment is inherently multithreaded. Therefore, your application initialization occurs in one thread, and the code that executes in response to the button press executes in a separate thread (the user interface thread).

With the C based IBM MQ MQI client, this would cause a problem, because there are limitations to the sharing of handles by multiple threads. IBM MQ classes for Java relaxes this constraint, allowing a queue manager object (and its associated queue, topic and process objects) to be shared by multiple threads.

The implementation of IBM MQ classes for Java ensures that, for a particular connection (MQQueueManager object instance), all access to the target IBM MQ queue manager is synchronized. A thread that wants to issue a call to a queue manager is blocked until all other calls in progress for that connection are complete. If you require simultaneous access to the same queue manager from multiple threads within your program, create a new MQQueueManager object for each thread that requires concurrent access. (This is equivalent to issuing a separate MQCONN call for each thread.)

Note: Instances of the class `com.ibm.mq.MQGetMessageOptions` must not be shared between threads which are requesting messages concurrently. Instances of this class are updated with data during

the corresponding MQGET request, which can result in unexpected consequences when multiple threads are operating concurrently on the same instance of the object.

Using channel exits in IBM MQ classes for Java

An overview of how to use channel exits in an application using the IBM MQ classes for Java.

The following topics describe how to write a channel exit in Java, how to assign it, and how to pass data to it. They then describe how to use channel exits written in C and how to use a sequence of channel exits.

Your application must have the correct security permission to load the channel exit class.

Creating a channel exit in IBM MQ classes for Java

You can provide your own channel exits by defining a Java class that implements an appropriate interface.

To implement an exit, you define a new Java class that implements the appropriate interface. Three exit interfaces are defined in the com.ibm.mq.exits package:

- WMQSendExit
- WMQReceiveExit
- WMQSecurityExit

Note: Channel exits are supported for client connections only; they are not supported for bindings connections. You cannot use a Java channel exit outside IBM MQ classes for Java, for example if you are using a client application written in C.

Any TLS encryption defined for a connection is performed *after* send and security exits have been invoked. Similarly, decryption is performed *before* receive and security exits are invoked.

The following sample defines a class that implements all three interfaces:

```
public class MyMQExits implements
WMQSendExit, WMQReceiveExit, WMQSecurityExit {
    // Default constructor
  public MyMQExits(){
  }
    // This method comes from the send exit interface
  public ByteBuffer channelSendExit(
MQCXP channelExitParms,
                                  MQCD channelDefinition,
                                  ByteBuffer agentBuffer)
  {
    // Fill in the body of the send exit here
  }
    // This method comes from the receive exit interface
  public ByteBuffer channelReceiveExit(
MQCXP channelExitParms,
                                     MQCD channelDefinition,
                                     ByteBuffer agentBuffer)
  {
    // Fill in the body of the receive exit here
  }
    // This method comes from the security exit interface
  public ByteBuffer channelSecurityExit(
MQCXP channelExitParms,
                                      MQCD channelDefinition,
                                      ByteBuffer agentBuffer)
  {
    // Fill in the body of the security exit here
  }
}
```

Each exit is passed an MQCXP object and an MQCD object. These objects represent the MQCXP and MQCD structures defined in the procedural interface.

Any exit class you write must have a constructor. This can be either the default constructor or one that takes a string argument. If it takes a string then the user data will be passed into the exit class when it is created. If the exit class contains both a default constructor and a single argument constructor, the single argument constructor has priority.

For the send and security exits, your exit code must return the data that you want to send to the server. For a receive exit, your exit code must return the modified data that you want IBM MQ to interpret.

The simplest possible exit body is:

```
{ return agentBuffer; }
```

Do not close the queue manager from within a channel exit.

Using existing channel exit classes

In versions of IBM MQ earlier than 7.0, you would implement these exits using the interfaces MQSendExit, MQReceiveExit, and MQSecurityExit, as in the following example. This method remains valid, but the new method is preferred for improved functionality and performance.

```
public class MyMQExits implements MQSendExit, MQReceiveExit, MQSecurityExit {
    // Default constructor
  public MyMQExits(){
  }
    // This method comes from the send exit
  public byte[] sendExit(MQChannelExit channelExitParms,
                         MQChannelDefinition channelDefParms,
                         byte agentBuffer[])
  {
    // Fill in the body of the send exit here
  }
    // This method comes from the receive exit
  public byte[] receiveExit(MQChannelExit channelExitParms,
                            MQChannelDefinition channelDefParms,
                            byte agentBuffer[])
  {
    // Fill in the body of the receive exit here
  }
    // This method comes from the security exit
  public byte[] securityExit(MQChannelExit channelExitParms,
                             MQChannelDefinition channelDefParms,
                             byte agentBuffer[])
  {
    // Fill in the body of the security exit here
  }
}
```

Assigning a channel exit in IBM MQ classes for Java
You can assign a channel exit using IBM MQ classes for Java.

There is no direct equivalent to the IBM MQ channel in IBM MQ classes for Java. Channel exits are assigned to an MQQueueManager. For example, having defined a class that implements the WMQSecurityExit interface, an application can use the security exit in one of four ways:

- By assigning an instance of the class to the MQEnvironment.channelSecurityExit field before creating an MQQueueManager object
- By setting the MQEnvironment.channelSecurityExit field to a string representing the security exit class before creating an MQQueueManager object
- By creating a key/value pair in the properties hashtable passed to MQQueueManager with a key of CMQC.SECURITY_EXIT_PROPERTY
- Using a client channel definition table (CCDT)

Any exit assigned by setting the MQEnvironment.channelSecurityExit field to a string, creating a key/value pair in the properties hashtable, or using a CCDT, must be written with a default constructor. An exit assigned as an instance of a class does not need a default constructor, depending on the application.

An application can use a send or a receive exit in a similar way. For example, the following code fragment shows you how to use the security, send, and receive exits that are implemented in the class MyMQExits, which was defined previously, using MQEnvironment:

```
MyMQExits myexits = new MyMQExits();
MQEnvironment.channelSecurityExit = myexits;
MQEnvironment.channelSendExit = myexits;
MQEnvironment.channelReceiveExit = myexits;
:
MQQueueManager jupiter = new MQQueueManager("JUPITER");
```

If more than one method is used to assign a channel exit, the order of precedence is as follows:

1. If the URL of a CCDT is passed to the MQQueueManager, the contents of the CCDT determine the channel exits to be used and any exit definitions in MQEnvironment or the properties hashtable are ignored.
2. If no CCDT URL is passed, exit definitions from MQEnvironment and the hashtable are merged
   - If the same exit type is defined in both MQEnvironment and the hashtable, the definition in the hashtable is used.
   - If equivalent old and new types of exit are specified (for example the sendExit field, which can only be used for the type of exit used in versions earlier than IBM WebSphere MQ 7.0, and the channelSendExit field, which can be used for any send exit), the new exit (channelSendExit) is used rather than the old exit.

If you have declared a channel exit as a string, you must enable IBM MQ to locate the channel exit program. You can do so in various ways, depending on the environment in which the application is running and on how the channel exit programs are packaged.

- For an application that is running in an application server, you must store the files in the directory shown in Table 58 on page 379 or packaged in JAR files referenced by **exitClasspath**.
- For an application that is not running in an application server, the following rules apply:
  - If your channel exit classes are packaged in separate JAR files, these JAR files must be included in the **exitClasspath**.
  - If your channel exit classes are not packaged in JAR files, the class files can be stored in the directory shown in Table 58 on page 379 or in any directory in the JVM system class path or **exitClasspath**.

The **exitClasspath** property can be specified in four ways. In order of priority, these ways are as follows:

1. The system property com.ibm.mq.exitClasspath (defined on the command line using the `-D` option)
2. The exitPath stanza of the `mqclient.ini` file
3. A hashtable entry with the key CMQC.EXIT_CLASSPATH_PROPERTY
4. The MQEnvironment variable **exitClasspath**

Separate multiple paths using the java.io.File.pathSeparator character.

Table 58. The directory for channel exit programs

Platform	Directory
AIX AIX Linux Linux	`/var/mqm/exits` (32-bit channel exit programs) `/var/mqm/exits64` (64-bit channel exit programs)
Windows	*install_data_dir*\exits

Note: *install_data_dir* is the directory that you chose for the IBM MQ data files during installation. The default directory is `C:\ProgramData\IBM\MQ`.

Passing data to channel exits in IBM MQ classes for Java
You can pass data to channel exits and return data from channel exits to your application.

The agentBuffer parameter

For a send exit, the *agentBuffer* parameter contains the data that is about to be sent. For a receive exit or a security exit, the *agentBuffer* parameter contains the data that has just been received. You do not need a length parameter, because the expression agentBuffer.limit() indicates the length of the array.

For the send and security exits, your exit code must return the data that you want to send to the server. For a receive exit, your exit code must return the modified data that you want IBM MQ to interpret.

The simplest possible exit body is:

```
{ return agentBuffer; }
```

Channel exits are called with a buffer that has a backing array. For best performance, the exit should return a buffer with a backing array.

User data

If an application connects to a queue manager by setting channelSecurityExit, channelSendExit, or channelReceiveExit, 32 bytes of user data can be passed to the appropriate channel exit class when it is called, using the channelSecurityExitUserData, channelSendExitUserData, or channelReceiveExitUserData fields. This user data is available to the channel exit class but is refreshed each time the exit is called. Any changes made to the user data in the channel exit will therefore be lost. If you want to make persistent changes to data in a channel exit, use the MQCXP exitUserArea. Data in this field is maintained between invocations of the exit.

If the application sets securityExit, sendExit, or receiveExit, no user data can be passed to these channel exit classes.

If an application uses a client channel definition table (CCDT) to connect to a queue manager, any user data specified in a client connection channel definition is passed to channel exit classes when they are called. For more information about using a client channel definition table, see “Using a client channel definition table with IBM MQ classes for Java” on page 363.

Using channel exits not written in Java with IBM MQ classes for Java
How to use channel exit programs written in C from a Java application.

In IBM MQ, you can specify the name of a channel exit program written in C as a String passed to the channelSecurityExit, channelSendExit, or channelReceiveExit fields in the MQEnvironment object or properties Hashtable. However, you cannot use a channel exit written in Java in an application written in another language.

Specify the exit program name in the format `library(function)` and ensure that the location of the exit program is specified as described in Path to exits.

For information about how to write a channel exit in C, see “Channel-exit programs for messaging channels” on page 926.

Using a sequence of channel send or receive exits in IBM MQ classes for Java
An IBM MQ classes for Java application can use a sequence of channel send or receive exits that are run in succession.

To use a sequence of send exits, an application can create either a List or a String containing the send exits. If a List is used, each element of the List can be any of the following:

- An instance of a user defined class that implements the WMQSendExit interface
- An instance of a user defined class that implements the MQSendExit interface (for a send exit written in Java)

- An instance of the MQExternalSendExit class (for a send exit not written in Java)
- An instance of the MQSendExitChain class
- An instance of the String class

A List cannot contain another List.

The application can use a sequence of receive exits in a similar manner.

If a String is used, it must consist of one or more comma-separated exit definitions, each of which can be the name of a Java class, or a C program in the format `library(function)`.

The application then assigns the List or String object to the MQEnvironment.channelSendExit field before creating an MQQueueManager object.

The context of information passed to exits is solely within the domain of the exits. For example, if a Java exit and a C exit are chained, the presence of the Java exit has no effect on the C exit.

Using exit chain classes

In versions earlier than IBM WebSphere MQ 7.0, two classes were provided to allow sequences of exits:

- MQSendExitChain, which implements the MQSendExit interface
- MQReceiveExitChain, which implements the MQReceiveExit interface

The use of these classes remains valid but the new method is preferred. Using the IBM MQ Classes for Java interfaces means that your application still has a dependency on `com.ibm.mq.jar` If the new set of interfaces in the `com.ibm.mq.exits` package are used there is no dependency on `com.ibm.mq.jar`.

To use a sequence of send exits, an application created a list of objects, where each object was one of the following:

- An instance of a user defined class that implements the MQSendExit interface (for a send exit written in Java)
- An instance of the MQExternalSendExit class (for a send exit not written in Java)
- An instance of the MQSendExitChain class

The application created an MQSendExitChain object by passing this list of objects as a parameter on the constructor. The application would then have assigned the MQSendExitChain object to the MQEnvironment.sendExit field before creating an MQQueueManager object.

Channel compression in IBM MQ classes for Java

Compressing the data that flows on a channel can improve the performance of the channel and reduce network traffic. IBM MQ classes for Java use the compression function built into IBM MQ.

Using function supplied with IBM MQ, you can compress the data that flows on message channels and MQI channels and, on either type of channel, you can compress header data and message data independently of each other. By default, no data is compressed on a channel. For a full description of channel compression, including how it is implemented in IBM MQ, see Data compression (COMPMSG) and Header compression (COMPHDR).

An IBM MQ classes for Java application specifies the techniques that can be used for compressing header or message data on a client connection by creating a java.util.Collection object. Each compression technique is an Integer object in the collection, and the order in which the application adds the compression techniques to the collection is the order in which the compression techniques are negotiated with the queue manager when the client connection starts. The application can then assign the collection to the hdrCompList field, for header data, or the msgCompList field, for message data, in the MQEnvironment class. When the application is ready, it can start the client connection by creating an MQQueueManager object.

The following code fragments illustrate the approach described. The first code fragment shows you how to implement header data compression:

```
Collection headerComp = new Vector();
headerComp.add(new Integer(CMQXC.MQCOMPRESS_SYSTEM));
:
MQEnvironment.hdrCompList = headerComp;
:
MQQueueManager qMgr = new MQQueueManager(QM);
```

The second code fragment shows you how to implement message data compression:

```
Collection msgComp = new Vector();
msgComp.add(new Integer(CMQXC.MQCOMPRESS_RLE));
msgComp.add(new Integer(CMQXC.MQCOMPRESS_ZLIBHIGH));
msgComp.add(new Integer(CMQXC.MQCOMPRESS_LZ4HIGH));
:
MQEnvironment.msgCompList = msgComp;
:
MQQueueManager qMgr = new MQQueueManager(QM);
```

In the second example, the compression techniques are negotiated in the order RLE, then ZLIBHIGH, when the client connection starts. The compression technique that is selected cannot be changed during the lifetime of the MQQueueManager object.

The compression techniques for header and message data that are supported by both the client and the queue manager on a client connection are passed to a channel exit as collections in the hdrCompList and msgCompList fields of an MQChannelDefinition object. The actual techniques that are currently being used for compressing header and message data on a client connection are passed to a channel exit in the CurHdrCompression and CurMsgCompression fields of an MQChannelExit object.

If compression is used on a client connection, the data is compressed before any channel send exits are processed and extracted after any channel receive exits are processed. The data passed to send and receive exits is therefore in a compressed state.

For more information about specifying compression techniques, and about which compression techniques are available, see Class com.ibm.mq.MQEnvironment and Interface com.ibm.mq.MQC.

Sharing a TCP/IP connection in IBM MQ classes for Java

Multiple instances of an MQI channel can be made to share a single TCP/IP connection.

In IBM MQ classes for Java, you use the MQEnvironment.sharingConversations variable to control the number of conversations that can share a single TCP/IP connection.

The SHARECNV attribute is a best effort approach to connection sharing. Therefore when a SHARECNV value greater than 0 is used with the IBM MQ classes for Java it is not guaranteed that a new connection request will always share an already established connection.

Connection pooling in IBM MQ classes for Java

IBM MQ classes for Java allows spare connections to be pooled for reuse.

IBM MQ classes for Java provides additional support for applications that deal with multiple connections to IBM MQ queue managers. When a connection is no longer required, instead of destroying it, it can be pooled and later reused. This can provide a substantial performance enhancement for applications and middleware that connect serially to arbitrary queue managers.

IBM MQ provides a default connection pool. Applications can activate or deactivate this connection pool by registering and deregistering tokens through the MQEnvironment class. If the pool is active when IBM MQ classes for Java constructs an MQQueueManager object, it searches this default pool and reuses any suitable connection. When an MQQueueManager.disconnect() call occurs, the underlying connection is returned to the pool.

Alternatively, applications can construct an MQSimpleConnectionManager connection pool for a particular use. Then, the application can either specify that pool during construction of an MQQueueManager object, or pass that pool to MQEnvironment for use as the default connection pool.

To prevent connections from using too much resource, you can limit the total number of connections that an MQSimpleConnectionManager object can handle, and you can limit the size of the connection pool. Setting limits is useful if there are conflicting demands for connections within a JVM.

By default, the getMaxConnections() method returns the value zero, which means that there is no limit to the number of connections that the MQSimpleConnectionManager object can handle. You can set a limit by using the setMaxConnections() method. If you set a limit and the limit is reached, a request for a further connection might cause an MQException to be thrown, with a reason code of MQRC_MAX_CONNS_LIMIT_REACHED.

Controlling the default connection pool in IBM MQ classes for Java
This example shows how to use the default connection pool.

Consider the following example application, MQApp1:

```
import com.ibm.mq.*;
public class MQApp1
{
      public static void main(String[] args) throws MQException
      {
         for (int i=0; i<args.length; i++) {
            MQQueueManager qmgr=new MQQueueManager(args[i]);
            :
            : (do something with qmgr)
            :
            qmgr.disconnect();
         }
      }
}
```

MQApp1 takes a list of local queue managers from the command line, connects to each in turn, and performs some operation. However, when the command line lists the same queue manager many times, it is more efficient to connect only once, and to reuse that connection many times.

IBM MQ classes for Java provides a default connection pool that you can use to do this. To enable the pool, use one of the MQEnvironment.addConnectionPoolToken() methods. To disable the pool, use MQEnvironment.removeConnectionPoolToken().

The following example application, MQApp2, is functionally identical to MQApp1, but connects only once to each queue manager.

```
import com.ibm.mq.*;
public class MQApp2
{
      public static void main(String[] args) throws MQException
      {
         MQPoolToken token=MQEnvironment.addConnectionPoolToken();

         for (int i=0; i<args.length; i++) {
            MQQueueManager qmgr=new MQQueueManager(args[i]);
            :
            : (do something with qmgr)
            :
            qmgr.disconnect();
         }

         MQEnvironment.removeConnectionPoolToken(token);

      }
}
```

The first bold line activates the default connection pool by registering an MQPoolToken object with MQEnvironment.

The MQQueueManager constructor now searches this pool for an appropriate connection and only creates a connection to the queue manager if it cannot find an existing one. The qmgr.disconnect() call returns the connection to the pool for later reuse. These API calls are the same as the sample application MQApp1.

The second highlighted line deactivates the default connection pool, which destroys any queue manager connections stored in the pool. This is important because otherwise the application would terminate with a number of live queue manager connections in the pool. This situation could cause errors that would appear in the queue manager logs.

If an application uses a client channel definition table (CCDT) to connect to a queue manager, the MQQueueManager constructor first searches the table for a suitable client connection channel definition. If one is found, the constructor searches the default connection pool for a connection that can be used for the channel. If the constructor cannot find a suitable connection in the pool, it then searches the client channel definition table for the next suitable client connection channel definition, and proceeds as described previously. If the constructor completes its search of the client channel definition table and fails to find any suitable connection in the pool, the constructor starts a second search of the table. During this search, the constructor tries to create a new connection for each suitable client connection channel definition in turn, and uses the first connection that it manages to create.

The default connection pool stores a maximum of ten unused connections, and keeps unused connections active for a maximum of five minutes. The application can alter this (for details, see “Supplying a different connection pool in IBM MQ classes for Java” on page 385).

Instead of using MQEnvironment to supply an MQPoolToken, the application can construct its own:

```
MQPoolToken token=new MQPoolToken();
MQEnvironment.addConnectionPoolToken(token);
```

Some applications or middleware vendors provide subclasses of MQPoolToken in order to pass information to a custom connection pool. They can be constructed and passed to addConnectionPoolToken() in this way so that extra information can be passed to the connection pool.

The default connection pool and multiple components in IBM MQ classes for Java
This example shows how to add or remove MQPoolTokens from a static set of registered MQPoolToken objects.

MQEnvironment holds a static set of registered MQPoolToken objects. To add or remove MQPoolTokens from this set, use the following methods:

- MQEnvironment.addConnectionPoolToken()
- MQEnvironment.removeConnectionPoolToken()

An application might consist of many components that exist independently and perform work using a queue manager. In such an application, each component should add an MQPoolToken to the MQEnvironment set for its lifetime.

For example, the example application MQApp3 creates ten threads and starts each one. Each thread registers its own MQPoolToken, waits for a length of time, then connects to the queue manager. After the thread disconnects, it removes its own MQPoolToken.

The default connection pool remains active while there is at least one token in the set of MQPoolTokens, so it will remain active for the duration of this application. The application does not need to keep a master object in overall control of the threads.

```
import com.ibm.mq.*;
public class MQApp3
{
      public static void main(String[] args)
      {
         for (int i=0; i<10; i++) {
            MQApp3_Thread thread=new MQApp3_Thread(i*60000);
            thread.start();
         }
      }
}

class MQApp3_Thread extends Thread
{
      long time;
```

```
      public MQApp3_Thread(long time)
      {
         this.time=time;
      }

      public synchronized void run()
      {
         MQPoolToken token=MQEnvironment.addConnectionPoolToken();
         try {
            wait(time);
            MQQueueManager qmgr=new MQQueueManager("my.qmgr.1");
            :
            : (do something with qmgr)
            :
            qmgr.disconnect();
         }
         catch (MQException mqe) {System.err.println("Error occurred!");}
         catch (InterruptedException ie) {}

         MQEnvironment.removeConnectionPoolToken(token);
      }
}
```

Supplying a different connection pool in IBM MQ classes for Java
This example shows how to use the class **com.ibm.mq.MQSimpleConnectionManager** to supply a different connection pool.

This class provides basic facilities for connection pooling, and applications can use this class to customize the behavior of the pool.

Once it is instantiated, an MQSimpleConnectionManager can be specified on the MQQueueManager constructor. The MQSimpleConnectionManager then manages the connection that underlies the constructed MQQueueManager. If the MQSimpleConnectionManager contains a suitable pooled connection, that connection is reused and returned to the MQSimpleConnectionManager after an MQQueueManager.disconnect() call.

The following code fragment demonstrates this behavior:

```
   MQSimpleConnectionManager myConnMan=new MQSimpleConnectionManager();
   myConnMan.setActive(MQSimpleConnectionManager.MODE_ACTIVE);
   MQQueueManager qmgr=new MQQueueManager("my.qmgr.1", myConnMan);
   :
   : (do something with qmgr)
   :
   qmgr.disconnect();

   MQQueueManager qmgr2=new MQQueueManager("my.qmgr.1", myConnMan);
   :
   : (do something with qmgr2)
   :
   qmgr2.disconnect();
   myConnMan.setActive(MQSimpleConnectionManager.MODE_INACTIVE);
```

The connection that is forged during the first MQQueueManager constructor is stored in myConnMan after the qmgr.disconnect() call. The connection is then reused during the second call to the MQQueueManager constructor.

The second line enables the MQSimpleConnectionManager. The last line disables MQSimpleConnectionManager, destroying any connections held in the pool. An MQSimpleConnectionManager is, by default, in MODE_AUTO, which is described later in this section.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes, or if there are more than ten unused connections in the pool. You can alter these values by calling MQSimpleConnectionManager.setTimeout().

You can also set up an MQSimpleConnectionManager for use as the default connection pool, to be used when no Connection Manager is supplied on the MQQueueManager constructor.

The following application demonstrates this:

```
import com.ibm.mq.*;
public class MQApp4
{
      public static void main(String []args)
      {
         MQSimpleConnectionManager myConnMan=new MQSimpleConnectionManager();
         myConnMan.setActive(MQSimpleConnectionManager.MODE_AUTO);
         myConnMan.setTimeout(3600000);
         myConnMan.setMaxConnections(75);
         myConnMan.setMaxUnusedConnections(50);
         MQEnvironment.setDefaultConnectionManager(myConnMan);
         MQApp3.main(args);
      }
}
```

The bold lines create and configure an MQSimpleConnectionManager object. The configuration does the following:

- Ends connections that are not used for an hour
- Limits the number of connections managed by myConnMan to 75
- Limits the number of unused connections in the pool to 50
- Sets MODE_AUTO, which is the default. This means that the pool is active only if it is the default connection manager, and there is at least one token in the set of MQPoolTokens held by MQEnvironment.

The new MQSimpleConnectionManager is then set as the default connection manager.

In the last line, the application calls MQApp3.main(). This runs a number of threads, where each thread uses IBM MQ independently. These threads use myConnMan when they forge connections.

JTA/JDBC coordination using IBM MQ classes for Java

IBM MQ classes for Java supports the MQQueueManager.begin() method, which allows IBM MQ to act as a coordinator for a database which provides a JDBC type 2 or JDBC type 4 compliant driver.

This support is not available on all platforms. To check which platforms support JDBC coordination, see System Requirements for IBM MQ.

To use the XA-JTA support, you must use the special JTA switch library. The method for using this library varies depending on whether you are using Windows or one of the other platforms.

Configuring JTA/JDBC coordination on Windows

The XA library is supplied as a DLL with a name of the format `jdbcxxx.dll`.

The supplied `jdbcora12.dll` provides compatibility with Oracle 12C, for an IBM MQ for Windows server installation.

On Windows systems, the XA library is supplied as a complete DLL. The name of this DLL is `jdbcxxx.dll` where `xxx` indicates the database for which the switch library has been compiled. This library is in the `java\lib\jdbc` or `java\lib64\jdbc` directory of your IBM MQ classes for Java installation. You must declare the XA library, also described as the switch load file, to the queue manager. Use the IBM MQ Explorer. Specify the details of the switch load file in the queue manager properties panel, under XA resource manager. You must only give the name of the library. For example:

For a Db2 database set the SwitchFile field to: `dbcdb2`

For an Oracle database set the SwitchFile field to: `jdbcora`

Notes:

1. Oracle 12C is supported by the IBM MQ classes for Java, only on IBM MQ for Windows.
2. The supported version of Oracle 12C is 12.1.0.1.0 Enterprise Edition and future fix packs.
3. Oracle 64-bit databases on 64- bit Windows require the 32-bit Oracle client.

4. Using the IBM MQ classes for Java, IBM MQ can act as a transaction coordinator. However it is not possible to participate in a JTA style transaction.

Configuring JTA/JDBC coordination on platforms other than Windows
Object files are supplied. Link the appropriate one using the supplied makefile, and declare it to the queue manager using the configuration file.

For each database management system, IBM MQ provides two object files. You must link one object file to create a 32-bit switch library, and link the other object file to create a 64-bit switch library. For Db2, the name of each object file is `jdbcdb2.o`, and for Oracle the name of each object file is `jdbcora.o`.

You must link each object file using the appropriate makefile supplied with IBM MQ. A switch library requires other libraries, which might be stored in different locations on different systems. However, a switch library cannot use the library path environment variable to locate these libraries because the switch library is loaded by the queue manager, which runs in a `setuid` environment. The supplied makefile therefore ensures that a switch library contains the fully qualified path names of these libraries.

To create a switch library, enter a **make** command with the following format. To create a 32-bit switch library, enter the command in the `/java/lib/jdbc` directory of your IBM MQ installation. To create a 64-bit switch library, enter the command in the `/java/lib64/jdbc` directory.

```
make DBMS
```

where *DBMS* is the database management system for which you are creating the switch library. The valid values are db2 for Db2 and `oracle` for Oracle.

Note:

- To run 32-bit applications, you must create both a 32-bit and a 64-bit switch library for each database management system that you are using. To run 64-bit applications, you need create only a 64-bit switch library. For Db2, the name of each switch library is `jdbcdb2` and, for Oracle, the name of each switch library is `jdbcora`. The makefiles ensure that 32-bit and 64-bit switch libraries are stored in different IBM MQ directories. A 32-bit switch library is stored in the `/java/lib/jdbc` directory, and a 64-bit switch library is stored in the `/java/lib64/jdbc` directory.
- Because you can install Oracle anywhere on a system, the makefiles use the **ORACLE_HOME** environment variable to locate where Oracle is installed.
- If IBM MQ is installed to a location other than the default location, alter the value of **MQ_INSTALLATION_PATH** in the makefile.

After you have created the switch libraries for Db2, Oracle, or both, you must declare them to your queue manager. If the queue manager configuration file (qm.ini) already contains `XAResourceManager` stanzas for Db2 or Oracle databases, you must replace the `SwitchFile` entry in each stanza by one of the following:

For a Db2 database

```
SwitchFile=jdbcdb2
```

For an Oracle database

```
SwitchFile=jdbcora
```

Do not specify the fully qualified path name of either the 32-bit or 64-bit switch library. Specify only the name of the library.

If the queue manager configuration file does not already contain `XAResourceManager` stanzas for Db2 or Oracle databases, or if you want to add additional `XAResourceManager` stanzas, see Administering IBM MQ for information about how to construct an `XAResourceManager` stanza. However, each `SwitchFile` entry in a new `XAResourceManager` stanza must be exactly as described previously for a Db2 or Oracle database. You must also include the entry `ThreadOfControl=PROCESS`.

After you have updated the queue manager configuration file, and made sure that all appropriate database environment variables have been set, you can restart the queue manager.

Using JTA/JDBC coordination
Code your API calls as in the supplied example.

The basic sequence of API calls for a user application is:

```
qMgr = new MQQueueManager("QM1")
Connection con = qMgr.getJDBCConnection( xads );
qMgr.begin()

< Perform MQ and DB operations to be grouped in a unit of work >

qMgr.commit() or qMgr.backout();
con.close()
qMgr.disconnect()
```

xads in the getJDBCConnection call is a database-specific implementation of the XADataSource interface, which defines the details of the database to connect to. See the documentation for your database to determine how to create an appropriate XADataSource object to pass into getJDBCConnection.

You must also update your class path with the appropriate database-specific jar files for performing JDBC work.

If you must connect to multiple databases, you must call getJDBCConnection several times to perform the transaction across several different connections.

There are two forms of the getJDBCConnection, reflecting the two forms of XADataSource.getXAConnection:

```
public java.sql.Connection getJDBCConnection(javax.sql.XADataSource xads)
  throws MQException, SQLException, Exception

public java.sql.Connection getJDBCConnection(XADataSource dataSource,
                                             String userid, String password)
  throws MQException, SQLException, Exception
```

These methods declare Exception in their throws clauses to avoid problems with the JVM verifier for customers who are not using the JTA functions. The actual exception thrown is javax.transaction.xa.XAException which requires the jta.jar file to be added to the class path for programs that did not previously require it.

To use the JTA/JDBC support, you must include the following statement in your application:

```
MQEnvironment.properties.put(CMQC.THREAD_AFFINITY_PROPERTY, new Boolean(true));
```

Known problems and limitations with JTA/JDBC coordination
Some of the problems and limitations of JTA/JDBC support depend on the database management system in use, for example, tested JDBC drivers behave differently when the database is shut down while an application is running. If the connection to the database that an application is using is broken, there are steps that the application can perform to reestablish a new connection to the queue manager and the database so that it can then use those new connections to perform the transactional work required.

Because the JTA/JDBC support makes calls to JDBC drivers, the implementation of those JDBC drivers can have a significant effect on the system behavior. In particular, tested JDBC drivers behave differently when the database is shut down while an application is running.

Important: Always avoid abruptly shutting down a database while there are applications that are holding open connections to it.

Note: An IBM MQ classes for Java application must connect by using bindings mode to make IBM MQ act as a database coordinator.

Multiple XAResourceManager stanzas
The use of more than one XAResourceManager stanza in a queue manager configuration file, `qm.ini`, is not supported. Any XAResourceManager stanza other than the first is ignored.

Db2

Sometimes Db2 returns a SQL0805N error. This problem can be resolved with the following CLP command:

```
DB2 bind @db2cli.lst blocking all grant public
```

For more information, refer to the Db2 documentation.

The XAResourceManager stanza must be configured to use ThreadOfControl=PROCESS. For Db2 8.1 and higher, this does not match the default thread of control setting for Db2, so toc=p must be specified in the XA Open String. An example XAResourceManager stanza for Db2 with JTA/JDBC coordination is as follows:

```
XAResourceManager:
     Name=jdbcdb2
     SwitchFile=jdbcdb2
     XAOpenString=uid=userid,db=dbalias,pwd=password,toc=p
     ThreadOfControl=PROCESS
```

This does not prevent the Java applications that use JTA/JDBC coordination from being multithreaded themselves.

Oracle
Calling the JDBC Connection.close() method after MQQueueManager.disconnect() generates an SQLException. Either call Connection.close() before MQQueueManager.disconnect(), or omit the call to Connection.close().

Handling issues with database connections

When an IBM MQ classes for Java application uses the JTA/JDBC support that is provided by IBM MQ, it typically performs the following steps:

1. Creates a new MQQueueManager object to represent a connection to the queue manager that will act as the transaction manager.
2. Constructs an XADataSource object that contains details about how to connect to the database that will be enlisted in the transaction.
3. Calls the method MQQueueManager.getJDBCConnection(XADataSource) passing in the XADataSource that was created previously. This causes the IBM MQ classes for Java to establish a connection to the database.
4. Calls the method MQQueueManager.begin() to start the XA transaction.
5. Performs the messaging and database work.
6. When all of the required work has been completed, calls the method MQQueueManager.commit(). This completes the XA transaction.
7. If a new XA transaction is required at this point, the application can repeat steps 4, 5 and 6.
8. When the application has finished, it should close the database connection that was created at step 3, and then call the method MQQueueManager.disconnect() to disconnect from the queue manager.

The IBM MQ classes for Java maintain an internal list of all of the database connections that have been created when an application calls MQQueueManager.getJDBCConnection(XADataSource). If a queue manager needs to communicate with the database during the processing of the XA transaction, the following processing takes place:

1. The queue manager calls into the IBM MQ classes for Java, passing in details of the XA call that needs to be passed to the database.

2. The IBM MQ classes for Java then look up the appropriate connection in the list, and then use that connection to flow the XA call to the database.

If the connection to the database is lost at any point during this processing, the application should:

1. Back out any existing work that was done under the transaction, by calling the method MQQueueManager.backout().
2. Close the database connection. This should cause the IBM MQ classes for Java to remove details of the broken database connection from its internal list.
3. Disconnect from the queue manager, by calling the method MQQueueManager.disconnect().
4. Establish a new connection to the queue manager, by constructing a new MQQueueManager object.
5. Create a new database connection, by calling the method MQQueueManager.getJDBCConnection(XADataSource).
6. Perform the transactional work again.

This allows the application to reestablish a new connection to the queue manager and the database, and then use those connections to perform the transactional work required.

Transport Layer Security (TLS) support in IBM MQ classes for Java

IBM MQ classes for Java client applications support TLS encryption. You require a JSSE provider to use TLS encryption.

IBM MQ classes for Java client applications using TRANSPORT(CLIENT) support TLS encryption. TLS provides communication encryption, authentication, and message integrity. It is typically used to secure communications between any two peers on the Internet or within an intranet.

IBM MQ classes for Java uses Java Secure Socket Extension (JSSE) to handle TLS encryption, and so requires a JSSE provider. JSE v1.4 JVMs have a JSSE provider built in. Details of how to manage and store certificates can vary from provider to provider. For information about this, refer to your JSSE provider's documentation.

This section assumes that your JSSE provider is correctly installed and configured, and that suitable certificates have been installed and made available to your JSSE provider.

If your IBM MQ classes for Java client application uses a client channel definition table (CCDT) to connect to a queue manager, see “Using a client channel definition table with IBM MQ classes for Java” on page 363.

Enabling TLS in IBM MQ classes for Java

To enable TLS, you specify a CipherSuite. There are two ways of specifying a CipherSuite.

TLS is supported only for client connections. To enable TLS, you must specify the CipherSuite to use when communicating with the queue manager, and this CipherSuite must match the CipherSpec set on the target channel. Additionally, the named CipherSuite must be supported by your JSSE provider. However, CipherSuites are distinct from CipherSpecs and so have different names. “TLS CipherSpecs and CipherSuites in IBM MQ classes for Java ” on page 394 contains a table mapping the CipherSpecs supported by IBM MQ to their equivalent CipherSuites as known to JSSE.

To enable TLS, specify the CipherSuite using the sslCipherSuite static member variable of MQEnvironment. The following example attaches to an SVRCONN channel named SECURE.SVRCONN.CHANNEL, which has been set up to require TLS with a CipherSpec of TLS_RSA_WITH_AES_128_CBC_SHA256:

```
MQEnvironment.hostname       = "your_hostname";
MQEnvironment.channel        = "SECURE.SVRCONN.CHANNEL";
MQEnvironment.sslCipherSuite = "SSL_RSA_WITH_AES_128_CBC_SHA256";
MQQueueManager qmgr = new MQQueueManager("your_Q_manager");
```

Although the channel has a CipherSpec of TLS_RSA_WITH_AES_128_CBC_SHA256, the Java application must specify a CipherSuite of SSL_RSA_WITH_AES_128_CBC_SHA256. See “TLS CipherSpecs and

CipherSuites in IBM MQ classes for Java ” on page 394 for a list of mappings between CipherSpecs and CipherSuites.

An application can also specify a CipherSuite by setting the environment property CMQC.SSL_CIPHER_SUITE_PROPERTY.

Alternatively, use the Client Channel Definition Table (CCDT). For more information, see “Using a client channel definition table with IBM MQ classes for Java” on page 363

If you require a client connection to use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS), an application can set the sslFipsRequired field in the MQEnvironment class to `true`. Alternatively, the application can set the environment property CMQC.SSL_FIPS_REQUIRED_PROPERTY. The default value is `false`, which means that a client connection can use any CipherSuite that is supported by IBM MQ.

If an application uses more than one client connection, the value of the sslFipsRequired field that is used when the application creates the first client connection determines the value that is used when the application creates any subsequent client connection. Therefore when the application creates a subsequent client connection, the value of the sslFipsRequired field is ignored. You must restart the application if you want to use a different value for the sslFipsRequired field.

To connect successfully using TLS, the JSSE truststore must be set up with certificate authority root certificates from which the certificate presented by the queue manager can be authenticated. Similarly, if SSLClientAuth on the SVRCONN channel has been set to MQSSL_CLIENT_AUTH_REQUIRED, the JSSE keystore must contain an identifying certificate that is trusted by the queue manager.

Related reference

Federal Information Processing Standards (FIPS) for AIX, Linux, and Windows

Using the distinguished name of the queue manager in IBM MQ classes for Java

The queue manager identifies itself using a TLS certificate, which contains a distinguished name (DN). An IBM MQ classes for Java client application can use this DN to ensure that it is communicating with the correct queue manager.

A DN pattern is specified using the sslPeerName variable of MQEnvironment. For example, setting:

```
MQEnvironment.sslPeerName = "CN=QMGR.*, OU=IBM, OU=WEBSPHERE";
```

allows the connection to succeed only if the queue manager presents a certificate with a Common Name beginning QMGR., and at least two Organizational Unit names, the first of which must be IBM and the second WebSphere.

If sslPeerName is set, connections succeed only if it is set to a valid pattern and the queue manager presents a matching certificate.

An application can also specify the distinguished name of the queue manager by setting the environment property CMQC.SSL_PEER_NAME_PROPERTY. For more information about distinguished names, see Distinguished names.

Using certificate revocation lists in IBM MQ classes for Java

Specify the certificate revocation lists to use through the java.security.cert.CertStore class. IBM MQ classes for Java then checks certificates against the specified CRL.

A certificate revocation list (CRL) is a set of certificates that have been revoked, either by the issuing certificate authority or by the local organization. CRLs are typically hosted on LDAP servers. With Java 2 v1.4, a CRL server can be specified at connect-time and the certificate presented by the queue manager is checked against the CRL before the connection is allowed. For more information about certificate revocation lists and IBM MQ, see Working with Certificate Revocation Lists and Authority Revocation Lists and Accessing CRLs and ARLs with IBM MQ classes for Java and IBM MQ classes for JMS.

Note: To use a CertStore successfully with a CRL hosted on an LDAP server, make sure that your Java Software Development Kit (SDK) is compatible with the CRL. Some SDKs require that the CRL conforms to RFC 2587, which defines a schema for LDAP v2. Most LDAP v3 servers use RFC 2256 instead.

The CRLs to use are specified through the java.security.cert.CertStore class. Refer to documentation on this class for full details of how to obtain instances of CertStore. To create a CertStore based on an LDAP server, first create an LDAPCertStoreParameters instance, initialized with the server and port settings to use. For example:

```
import java.security.cert.*;
CertStoreParameters csp = new LDAPCertStoreParameters("crl_server", 389);
```

Having created a CertStoreParameters instance, use the static constructor on CertStore to create a CertStore of type LDAP:

```
CertStore cs = CertStore.getInstance("LDAP", csp);
```

Other CertStore types (for example, Collection) are also supported. Commonly there are several CRL servers set up with identical CRL information to give redundancy. When you have a CertStore object for each of these CRL servers, place them all in a suitable Collection. The following example shows the CertStore objects placed in an ArrayList:

```
import java.util.ArrayList;
Collection crls = new ArrayList();
crls.add(cs);
```

This Collection can be set into the MQEnvironment static variable, sslCertStores, before connecting to enable CRL checking:

```
MQEnvironment.sslCertStores = crls;
```

The certificate presented by the queue manager when a connection is being set up is validated as follows:

1. The first CertStore object in the Collection identified by sslCertStores is used to identify a CRL server.
2. An attempt is made to contact the CRL server.
3. If the attempt is successful, the server is searched for a match for the certificate.
   a. If the certificate is found to be revoked, the search process is over and the connection request fails with reason code MQRC_SSL_CERTIFICATE_REVOKED.
   b. If the certificate is not found, the search process is over and the connection is allowed to proceed.
4. If the attempt to contact the server is unsuccessful, the next CertStore object is used to identify a CRL server and the process repeats from step 2.

   If this was the last CertStore in the Collection, or if the Collection contains no CertStore objects, the search process failed, and the connection request fails with reason code MQRC_SSL_CERT_STORE_ERROR.

The Collection object determines the order in which CertStores are used.

The Collection of CertStores can also be set using the CMQC.SSL_CERT_STORE_PROPERTY. As a convenience, this property also allows a single CertStore to be specified without being a member of a Collection.

If sslCertStores is set to null, no CRL checking is performed. This property is ignored if sslCipherSuite is not set.

Renegotiating the secret key in IBM MQ classes for Java
An IBM MQ classes for Java client application can control when the secret key that is used for encryption on a client connection is renegotiated, in terms of the total number of bytes sent and received.

The application can do this in either of the following ways: If the application uses more than one of these ways, the usual precedence rules apply.

- By setting the sslResetCount field in the MQEnvironment class.

- By setting the environment property MQC.SSL_RESET_COUNT_PROPERTY in a Hashtable object. The application then assigns the hashtable to the `properties` field in the MQEnvironment class, or passes the hashtable to an MQQueueManager object on its constructor.

The value of the sslResetCount field or environment property MQC.SSL_RESET_COUNT_PROPERTY represents the total number of bytes sent and received by the IBM MQ classes for Java client code before the secret key is renegotiated. The number of bytes sent is the number before encryption, and the number of bytes received is the number after decryption. The number of bytes also includes control information sent and received by the IBM MQ classes for Java client.

If the reset count is zero, which is the default value, the secret key is never renegotiated. The reset count is ignored if no CipherSuite is specified.

Supplying a customized SSLSocketFactory in IBM MQ classes for Java
If you use a customized JSSE Socket Factory, set the MQEnvironment.sslSocketFactory to the customized factory object. Details vary between different JSSE implementations.

Different JSSE implementations can provide different features. For example, a specialized JSSE implementation might allow configuration of a particular model of encryption hardware. Additionally, some JSSE providers allow customization of keystores and truststores by program, or allow the choice of identity certificate from the keystore to be altered. In JSSE, all these customizations are abstracted into a factory class, javax.net.ssl.SSLSocketFactory.

See your JSSE documentation for details of how to create a customized SSLSocketFactory implementation. The details vary from provider to provider, but a typical sequence of steps might be:

1. Create an SSLContext object using a static method on SSLContext
2. Initialize this SSLContext with appropriate KeyManager and TrustManager implementations (created from their own factory classes)
3. Create an SSLSocketFactory from the SSLContext

When you have an SSLSocketFactory object, set the MQEnvironment.sslSocketFactory to the customized factory object. For example:

```
javax.net.ssl.SSLSocketFactory sf = sslContext.getSocketFactory();
MQEnvironment.sslSocketFactory = sf;
```

IBM MQ classes for Java use this SSLSocketFactory to connect to the IBM MQ queue manager. This property can also be set using the CMQC.SSL_SOCKET_FACTORY_PROPERTY. If sslSocketFactory is set to null, the default SSLSocketFactory of the JVM is used. This property is ignored if sslCipherSuite is not set.

When you use custom SSLSocketFactories, consider the effect of TCP/IP connection sharing. If connection sharing is possible then a new socket is not requested of the SSLSocketFactory supplied, even if the socket produced would be different in some way in the context of a subsequent connection request. For example, if a different client certificate is to be presented on a subsequent connection, then connection sharing must not be allowed.

Making changes to the JSSE keystore or truststore in IBM MQ classes for Java
If you change the JSSE keystore or truststore, you must perform certain actions for the changes to take effect.

If you change the contents of the JSSE keystore or truststore, or change the location of the keystore or truststore file, IBM MQ classes for Java applications that are running at the time do not automatically pick up the changes. For the changes to take effect, the following actions must be performed:

- The applications must close all their connections, and destroy any unused connections in connection pools.
- If your JSSE provider caches information from the keystore and truststore, this information must be refreshed.

After these actions have been performed, the applications can then re-create their connections.

Depending on how you design your applications, and on the function provided by your JSSE provider, it might be possible to perform these actions without stopping and restarting your applications. However, stopping and restarting the applications might be the simplest solution.

Error handling when using TLS with IBM MQ classes for Java
A number of reason codes can be issued by IBM MQ classes for Java when connecting to a queue manager using TLS.

These are explained in the following list:

MQRC_SSL_NOT_ALLOWED
The sslCipherSuite property was set, but bindings connect was used. Only client connect supports TLS.

MQRC_JSSE_ERROR
The JSSE provider reported an error that could not be handled by IBM MQ. This could be caused by a configuration problem with JSSE, or because the certificate presented by the queue manager could not be validated. The exception produced by JSSE can be retrieved using the getCause() method on MQException.

MQRC_SSL_INITIALIZATION_ERROR
An MQCONN or MQCONNX call was issued with TLS configuration options specified, but an error occurred during the initialization of the TLS environment.

MQRC_SSL_PEER_NAME_MISMATCH
The DN pattern specified in the sslPeerName property did not match the DN presented by the queue manager.

MQRC_SSL_PEER_NAME_ERROR
The DN pattern specified in the sslPeerName property was not valid.

MQRC_UNSUPPORTED_CIPHER_SUITE
The CipherSuite named in sslCipherSuite was not recognized by the JSSE provider. A full list of CipherSuites supported by the JSSE provider can be obtained by a program using the SSLSocketFactory.getSupportedCipherSuites() method. A list of CipherSuites that can be used to communicate with IBM MQ can be found in "TLS CipherSpecs and CipherSuites in IBM MQ classes for Java " on page 394.

MQRC_SSL_CERTIFICATE_REVOKED
The certificate presented by the queue manager was found in a CRL specified with the sslCertStores property. Update the queue manager to use trusted certificates.

MQRC_SSL_CERT_STORE_ERROR
None of the supplied CertStores could be searched for the certificate presented by the queue manager. The MQException.getCause() method returns the error that occurred while searching the first CertStore attempted. If the causal exception is NoSuchElementException, ClassCastException, or NullPointerException, check that the Collection specified on the sslCertStores property contains at least one valid CertStore object.

TLS CipherSpecs and CipherSuites in IBM MQ classes for Java
The ability of IBM MQ classes for Java applications to establish connections to a queue manager, depends on the CipherSpec specified at the server end of the MQI channel and the CipherSuite specified at the client end.

The following table lists the CipherSpecs supported by IBM MQ and their equivalent CipherSuites.

Deprecated You should review the topic Deprecated CipherSpecs to see if any of the CipherSpecs, listed in the following table, have been deprecated by IBM MQ and, if so, at which update the CipherSpec was deprecated.

Important: The CipherSuites listed are those supported by the IBM Java Runtime Environment (JRE) supplied with IBM MQ. The CipherSuites that are listed include those supported by the Oracle Java JRE. For more information about configuring your application to use an Oracle Java JRE, see Configuring your application to use IBM Java or Oracle Java CipherSuite mappings.

The table also indicates the protocol that is used for the communication, and whether or not the CipherSuite conforms to the FIPS 140-2 standard.

Note: On AIX, Linux, and Windows, IBM MQ provides FIPS 140-2 compliance through the IBM Crypto for C (ICC) cryptographic module. The certificate for this module has been moved to the Historical status. Customers should view the IBM Crypto for C (ICC) certificate and be aware of any advice provided by NIST. A replacement FIPS 140-3 module is currently in progress and its status can be viewed by searching for it in the NIST CMVP modules in process list.

The IBM MQ Operator 3.2.0 and queue manager container image 9.4.0.0 onwards are based on UBI 9. FIPS 140-3 compliance is currently pending and its status can be viewed by searching for "Red Hat Enterprise Linux 9 - OpenSSL FIPS Provider" in the NIST CMVP modules in process list.

Ciphersuites denoted as FIPS 140-2 compliant can be used if the application has not been configured to enforce FIPS 140-2 compliance, but if FIPS 140-2 compliance has been configured for the application (see the following notes on configuration) only those CipherSuites which are marked as FIPS 140-2 compatible can be configured; attempting to use other CipherSuites results in an error.

Note: Each JRE can have multiple cryptographic security providers, each of which can contribute an implementation of the same CipherSuite. However, not all security providers are FIPS 140-2 certified. If FIPS 140-2 compliance is not enforced for an application then it is possible that an uncertified implementation of the CipherSuite might be used. Uncertified implementations might not operate in compliance with FIPS 140-2, even if the CipherSuite theoretically meets the minimum security level required by the standard. See the following notes for more information about configuring FIPS 140-2 enforcement in IBM MQ Java applications.

For more information about FIPS 140-2 and Suite-B compliance for CipherSpecs and CipherSuites, see Specifying CipherSpecs. You might also need to be aware of information that concerns US Federal Information Processing Standards.

To use the full set of CipherSuites and to operate with certified FIPS 140-2 and/or Suite-B compliance, a suitable JRE is required. IBM Java 7 Service Refresh 4 Fix Pack 2 or a higher level of IBM JRE provides the appropriate support for the TLS 1.2 CipherSuites listed in Table 59 on page 396.

To be able to use TLS 1.3 Ciphers, the JRE running your application must support TLS 1.3.

Note: To use some CipherSuites, the 'unrestricted' policy files need to be configured in the JRE. For more details of how policy files are set up in an SDK or JRE, see the *IBM SDK Policy files* topic in the *Security Reference for IBM SDK, Java Technology Edition* for the version you are using.

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_3DES_EDE_CBC_SHA256	SSL_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_128_CBC_SHA256	SSL_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_128_GCM_SHA256	SSL_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_256_CBC_SHA384	SSL_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_AES_256_GCM_SHA384	SSL_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS 1.2	yes
ECDHE_ECDSA_NULL_SHA256	SSL_ECDHE_ECDSA_WITH_NULL_SHA	TLS_ECDHE_ECDSA_WITH_NULL_SHA	TLS 1.2	no

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_ECDSA_RC4_128_SHA256	SSL_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS 1.2	no
ECDHE_RSA_3DES_EDE_CBC_SHA256	SSL_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_128_CBC_SHA256	SSL_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_128_GCM_SHA256	SSL_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_256_CBC_SHA384	SSL_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS 1.2	yes

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_AES_256_GCM_SHA384	SSL_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2	yes
ECDHE_RSA_NULL_SHA256	SSL_ECDHE_RSA_WITH_NULL_SHA	TLS_ECDHE_RSA_WITH_NULL_SHA	TLS 1.2	no

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ECDHE_RSA_RC4_128_SHA256	SSL_ECDHE_RSA_WITH_RC4_128_SHA	TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLS 1.2	no
TLS_RSA_WITH_3DES_EDE_CBC_SHA "2" on page 414	SSL_RSA_WITH_3DES_EDE_CBC_SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS 1.0	no "4" on page 414

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_AES_128_CBC_SHA	SSL_RSA_WITH_AES_128_CBC_SHA	TLS_RSA_WITH_AES_128_CBC_SHA	TLS 1.0	no "4" on page 414
TLS_RSA_WITH_AES_128_CBC_SHA256	SSL_RSA_WITH_AES_128_CBC_SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2	no "4" on page 414

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_AES_128_GCM_SHA256	SSL_RSA_WITH_AES_128_GCM_SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2	no "4" on page 414
TLS_RSA_WITH_AES_256_CBC_SHA	SSL_RSA_WITH_AES_256_CBC_SHA	TLS_RSA_WITH_AES_256_CBC_SHA	TLS 1.0	no "4" on page 414

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_AES_256_CBC_SHA256	SSL_RSA_WITH_AES_256_CBC_SHA256	TLS_RSA_WITH_AES_256_CBC_SHA256	TLS 1.2	no "4" on page 414
TLS_RSA_WITH_AES_256_GCM_SHA384	SSL_RSA_WITH_AES_256_GCM_SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2	no "4" on page 414

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA	TLS 1.0	no
TLS_RSA_WITH_NULL_SHA256	SSL_RSA_WITH_NULL_SHA256	TLS_RSA_WITH_NULL_SHA256	TLS 1.2	no
TLS_RSA_WITH_RC4_128_SHA256	SSL_RSA_WITH_RC4_128_SHA	SSL_RSA_WITH_RC4_128_SHA	TLS 1.2	no

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
ANY_TLS12	*TLS12	*TLS12	TLS 1.2	yes
TLS_AES_128_GCM_SHA256 "3" on page 414	TLS_AES_128_GCM_SHA256	TLS_AES_128_GCM_SHA256	TLS V1.3	no
TLS_AES_256_GCM_SHA384 "3" on page 414	TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384	TLS V1.3	no

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec "1" on page 414	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_CHACHA20_POLY1305_SHA256 "3" on page 414	TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256	TLS V1.3	no
TLS_AES_128_CCM_SHA256 "3" on page 414	TLS_AES_128_CCM_SHA256	TLS_AES_128_CCM_SHA256	TLS V1.3	no

Table 59. CipherSpecs supported by IBM MQ and their equivalent CipherSuites (continued)

CipherSpec **"1" on page 414**	**Equivalent CipherSuite (IBM JRE)**	**Equivalent CipherSuite (Oracle JRE)**	**Protocol**	**FIPS 140-2 compatible**
TLS_AES_128_CCM_8_SHA256 "3" on page 414	TLS_AES_128_CCM_8_SHA256	TLS_AES_128_CCM_8_SHA256	TLS V1.3	no
ANY "3" on page 414	*ANY	*ANY	Multiple	no
ANY_TLS13 "3" on page 414	*TLS13	*TLS13	TLS V13	no
ANY_TLS12_OR_HIGHER "3" on page 414	*TLS12ORHIGHER	*TLS12ORHIGHER	TLS 1.2 and above	no
ANY_TLS13_OR_HIGHER "3" on page 414	*TLS13ORHIGHER	*TLS13ORHIGHER	TLS 1.3 and above	no

Notes:

1. This is the value configured on a channel in IBM MQ, including in a CCDT (binary or JSON).
2. Deprecated CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.
3. To be able to use TLS v1.3 Ciphers, the Java runtime environment (JRE) running your application must support TLS v1.3.
4. V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, the IBM Java 8 JRE removes support for RSA key exchange when operating in FIPS mode.

Configuring Ciphersuites and FIPS-compliance in an IBM MQ classes for Java application

- An application that uses IBM MQ classes for Java can use either of two methods to set the CipherSuite for a connection:
  - Set the sslCipherSuite field in the MQEnvironment class to the CipherSuite name.
  - Set the property CMQC.SSL_CIPHER_SUITE_PROPERTY in the properties hashtable passed to the MQQueueManager constructor to the CipherSuite name.
- An application that uses IBM MQ classes for Java can use either of two methods to enforce FIPS 140-2 compliance:
  - Set the sslFipsRequired field to true in the MQEnvironment class.
  - Set the property CMQC.SSL_FIPS_REQUIRED_PROPERTYin the properties hash table passed to the MQQueueManager constructor to true.

Configuring your application to use IBM Java or Oracle Java CipherSuite mappings

V 9.4.0 From IBM MQ 9.4.0, a Cipher can be defined as either the CipherSpec or CipherSuite name and is handled correctly by IBM MQ.

Note: Removed The Java System Property `com.ibm.mq.cfg.useIBMCipherMappings`, which controlled which mappings were used in earlier versions of IBM MQ, is no longer needed and is removed from the product at IBM MQ 9.4.0.

Interoperability limitations

Certain CipherSuites might be compatible with more than one IBM MQ CipherSpec, depending on the protocol in use. However, only the CipherSuite/CipherSpec combination that uses the TLS version specified in Table 1 is supported. Attempting to use the unsupported combinations of CipherSuites and CipherSpecs will fail with an appropriate exception. Installations using any of these CipherSuite/CipherSpec combinations should move to a supported combination.

The following table shows the CipherSuites to which this limitation applies.

Table 60. CipherSuites and their supported and unsupported CipherSpecs

CipherSuite	Supported TLS CipherSpec	Unsupported SSL CipherSpec
SSL_RSA_WITH_3DES_EDE_CBC_SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA "1" on page 415	TRIPLE_DES_SHA_US
SSL_RSA_WITH_DES_CBC_SHA	TLS_RSA_WITH_DES_CBC_SHA	DES_SHA_EXPORT
SSL_RSA_WITH_RC4_128_SHA	TLS_RSA_WITH_RC4_128_SHA256	RC4_SHA_US

Note:

1. Deprecated This CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.

Running IBM MQ classes for Java applications

If you write an application (a class that contains a main() method), using either the client or the bindings mode, run your program using the Java interpreter.

Use the command:

```
java -Djava.library.path= library_path MyClass
```

where *library_path* is the path to the IBM MQ classes for Java libraries. for more information, see "IBM MQ classes for Java libraries" on page 345.

Related tasks

Tracing IBM MQ classes for Java applications

Tracing the IBM MQ Resource Adapter

IBM MQ classes for Java environment-dependent behavior

IBM MQ classes for Java allow you to create applications that can run against different versions of IBM MQ. This collection of topics describes the behavior of the Java classes dependent on these different versions.

IBM MQ classes for Java provides a core of classes, which provide consistent function and behavior in all the environments. Features outside this core depend on the capability of the queue manager to which the application is connected.

Except where noted here, the behavior exhibited is as described in the MQI application reference appropriate to the queue manager.

Core classes in IBM MQ classes for Java

IBM MQ classes for Java contains a core set of classes, which can be used in all environments.

The following set of classes are considered core classes, and can be used in all environments with only the minor variations listed in "Restrictions and variations for core classes of IBM MQ classes for Java" on page 416.

- MQEnvironment
- MQException
- MQGetMessageOptions

  Excluding:

  - MatchOptions
  - GroupStatus
  - SegmentStatus
  - Segmentation
- MQManagedObject

  Excluding:

  - inquire()
  - set()
- MQMessage

Excluding:

- groupId
- messageFlags
- messageSequenceNumber
- offset
- originalLength

- MQPoolServices
- MQPoolServicesEvent
- MQPoolServicesEventListener
- MQPoolToken
- MQPutMessageOptions

  Excluding:

  - knownDestCount
  - unknownDestCount
  - invalidDestCount
  - recordFields
- MQProcess
- MQQueue
- MQQueueManager

  Excluding:

  - begin()
  - accessDistributionList()
- MQSimpleConnectionManager
- MQTopic
- MQC

Note:

1. Some constants are not included in the core (see "Restrictions and variations for core classes of IBM MQ classes for Java" on page 416 for details); do not use them in completely portable programs.
2. Some platforms do not support all connection modes. On these platforms, you can use only the core classes and options that relate to the supported modes.

Restrictions and variations for core classes of IBM MQ classes for Java

The core classes generally behave consistently across all environments, even if the equivalent MQI calls normally have environment differences. The behavior is as if a AIX, Linux, or Windows queue manager is used, except for the following minor restrictions and variations.

Restrictions for MQGMO_* values in IBM MQ classes for Java

Certain MQGMO_* values are not supported by all queue managers.

Use of the following MQGMO_* values might result in an MQException being thrown from an MQQueue.get():

MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_MARK_SKIP_BACKOUT
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_LOCK
MQGMO_UNLOCK
MQGMO_LOGICAL_ORDER

MQGMO_COMPLETE_MESSAGE
MQGMO_ALL_MSGS_AVAILABLE
MQGMO_ALL_SEGMENTS_AVAILABLE
MQGMO_UNMARKED_BROWSE_MSG
MQGMO_MARK_BROWSE_HANDLE
MQGMO_MARK_BROWSE_CO_OP
MQGMO_UNMARK_BROWSE_HANDLE
MQGMO_UNMARK_BROWSE_CO_OP

Additionally, MQGMO_SET_SIGNAL is not supported when used from Java.

Restrictions for MQPMRF_* values in IBM MQ classes for Java
These are used only when putting messages to a distribution list, and are supported only by queue managers supporting distribution lists. For example, z/OS queue managers do not support distribution lists.

Restrictions for MQPMO_* values in IBM MQ classes for Java
Certain MQPMO_* values are not supported by all queue managers

Use of the following MQPMO_* values might result in an MQException being thrown from an MQQueue.put() or an MQQueueManager.put():

MQPMO_LOGICAL_ORDER
MQPMO_NEW_CORREL_ID
MQPMO_NEW_MESSAGE_ID
MQPMO_RESOLVE_LOCAL_Q

Restrictions and variations for MQCNO_* values in IBM MQ classes for Java
Certain MQCNO_* values are not supported.

- Automatic client reconnect is not supported by the IBM MQ classes for Java. Whatever value MQCNO_RECONNECT_* you set, the connection continues to behave as if you set MQCNO_RECONNECT_DISABLED.
- MQCNO_FASTPATH is ignored on queue managers that do not support MQCNO_FASTPATH. It is also ignored by client connections.

Restrictions for MQRO_* values in IBM MQ classes for Java
The following report options can be set.

MQRO_EXCEPTION_WITH_FULL_DATA
MQRO_EXPIRATION_WITH_FULL_DATA
MQRO_COA_WITH_FULL_DATA
MQRO_COD_WITH_FULL_DATA
MQRO_DISCARD_MSG
MQRO_PASS_DISCARD_AND_EXPIRY

For more information see Report.

Miscellaneous differences between IBM MQ classes for Java on z/OS and other platforms
IBM MQ for z/OS behaves differently from IBM MQ on other platforms in some areas.

BackoutCount
A z/OS queue manager returns a maximum BackoutCount of 255, even if the message has been backed out more than 255 times.

Default dynamic queue prefix
When connected to a z/OS queue manager using a bindings connection, the default dynamic queue prefix is CSQ.*. Otherwise, the default dynamic queue prefix is AMQ.*.

MQQueueManager constructor

Client connect is not supported on z/OS. Attempting to connect with client options results in an MQException with MQCC_FAILED and MQRC_ENVIRONMENT_ERROR.

The MQQueueManager constructor might also fail with MQRC_CHAR_CONVERSION_ERROR (if it fails to initialize conversion between the IBM-1047 and ISO8859-1 code pages), or MQRC_UCS2_CONVERSION_ERROR (if it fails to initialize conversion between the queue manager's code page and Unicode). If your application fails with one of these reason codes, ensure that the National Language Resources component of Language Environment is installed, and ensure that the correct conversion tables are available.

Conversion tables for Unicode are installed as part of the z/OS C/C++ optional feature. See the *z/OS C/C++ Programming Guide*, SC09-4765, for more information about enabling UCS-2 conversions.

Features outside the core classes of IBM MQ classes for Java

IBM MQ classes for Java contain certain functions that are specifically designed to use API extensions that are not supported by all queue managers. This collection of topics describes how they behave when using a queue manager that does not support them.

Variations in the MQQueueManager constructor option

Some of the MQQueueManager constructors include an optional integer argument. Some values of this argument are not accepted on all platforms.

Where an MQQueueManager constructor include an optional integer argument, it maps onto the MQCNO options field of the MQI, and is used to switch between normal and fast path connection. This extended form of the constructor is accepted in all environments, if the only options used are MQCNO_STANDARD_BINDING or MQCNO_FASTPATH_BINDING. Any other options cause the constructor to fail with MQRC_OPTIONS_ERROR. The fast path option CMQC.MQCNO_FASTPATH_BINDING is honored only with a bindings connection to a queue manager that supports it. In other environments, it is ignored.

Restrictions on the MQQueueManager.begin() method

This method can be used only against an IBM MQ queue manager on AIX, Linux, and Windows systems in bindings mode. Otherwise, it fails with MQRC_ENVIRONMENT_ERROR.

See "JTA/JDBC coordination using IBM MQ classes for Java" on page 386 for more details.

Variations in the MQGetMessageOptions fields

Some queue managers do not support the Version 2 MQGMO structure, so you must set some fields to their default values.

When using a queue manager that does not support the Version 2 MQGMO structure, leave the following fields set to their default values:

GroupStatus
SegmentStatus
Segmentation

Also, the MatchOptions field supports only MQMO_MATCH_MSG_ID and MQMO_MATCH_CORREL_ID. If you put unsupported values into these fields, the subsequent MQDestination.get() fails with MQRC_GMO_ERROR. If the queue manager does not support the Version 2 MQGMO structure, these fields are not updated after a successful MQDestination.get().

Restrictions in distribution lists in IBM MQ classes for Java

Not all queue managers allow you to open an MQDistributionList.

The following classes are used to create distribution lists:

MQDistributionList
MQDistributionListItem
MQMessageTracker

You can create and populate MQDistributionLists and MQDistributionListItems in any environment, but not all queue managers allow you to open an MQDistributionList. In particular, z/OS queue managers do not support distribution lists. Attempting to open an MQDistributionList when using such a queue manager results in MQRC_OD_ERROR.

Variations in MQPutMessageOptions fields
If a queue manager does not support distribution lists, certain MQPMO fields are treated differently.

Four fields in the MQPMO are rendered as the following member variables in the MQPutMessageOptions class:

knownDestCount
unknownDestCount
invalidDestCount
recordFields

These fields are primarily intended for use with distribution lists. However, a queue manager that supports distribution lists also fills in the DestCount fields after an MQPUT to a single queue. For example, if the queue resolves to a local queue, knownDestCount is set to 1 and the other two count fields are set to 0.

If the queue manager does not support distribution lists, these values are simulated as follows:

- If the put() succeeds, unknownDestCount is set to 1, and the others are set to 0.
- If the put() fails, invalidDestCount is set to 1, and the others are set to 0.

The recordFields variable is used with distribution lists. A value can be written into recordFields at any time, regardless of the environment. It is ignored if the MQPutMessageOptions object is used on a subsequent MQDestination.put() or MQQueueManager.put(), rather than MQDistributionList.put().

Restrictions in MQMD fields with IBM MQ classes for Java
Certain MQMD fields concerned with message segmentation should be left at their default value when using a queue manager that does not support segmentation.

The following MQMD fields are largely concerned with message segmentation:

GroupId
MsgSeqNumber
Offset
MsgFlags
OriginalLength

If an application sets any of these MQMD fields to values other than their defaults, and then does a put() or get() on a queue manager that does not support these, the put() or get() raises an MQException with MQRC_MD_ERROR. A successful put() or get() with such a queue manager always leaves the MQMD fields set to their default values. Do not send a grouped or segmented message to a Java application that runs against a queue manager that does not support message grouping and segmentation.

If a Java application attempts to get() a message from a queue manager that does not support these fields, and the physical message to be retrieved is part of a group of segmented messages (that is, it has non-default values for the MQMD fields), it is retrieved without error. However, the MQMD fields in the MQMessage are not updated, the MQMessage format property is set to MQFMT_MD_EXTENSION, and the true message data is prefixed with an MQMDE structure that contains the values for the new fields.

Restrictions for IBM MQ classes for Java under CICS Transaction Server
In the CICS Transaction Server for z/OS environment, only the main (first) thread is allowed to issue CICS or IBM MQ calls.

Note, that IBM MQ JMS classes are not supported for use in a CICS Java application.

It is therefore not possible to share MQQueueManager or MQQueue objects between threads in this environment, or to create a new MQQueueManager on a child thread.

z/OS "Miscellaneous differences between IBM MQ classes for Java on z/OS and other platforms" on page 417 identifies some restrictions and variations that apply to the IBM MQ classes for Java when running against a z/OS queue manager. Additionally, when running under CICS, the transaction control methods on MQQueueManager are not supported. Instead of issuing MQQueueManager.commit() or MQQueueManager.backout(), applications use the JCICS task synchronization methods, Task.commit() and Task.rollback(). The Task class is supplied by JCICS in the com.ibm.cics.server package.

Using the IBM MQ resource adapter

The resource adapter allows applications that are running in an application server to access IBM MQ resources. It supports inbound and outbound communication.

What the resource adapter contains

From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. In addition to the resource adapter that supports Java EE and JMS 2.0, IBM MQ 9.3.0 and later provide a resource adapter that supports Jakarta Messaging.

JMS 3.0 **IBM MQ resource adapter for Jakarta Messaging**

The Jakarta Connectors Architecture provides a standard way of connecting applications that are running in a Jakarta EE environment to an Enterprise Information System (EIS) such as IBM MQ or Db2. The IBM MQ resource adapter for Jakarta Messaging implements the Jakarta Connectors 2.0.0 interfaces and contains the IBM MQ classes for Jakarta Messaging. It allows Jakarta Messaging applications and message driven beans (MDBs) running in an application server, to access the resources of an IBM MQ queue manager. The resource adapter supports both the point-to-point domain and the publish/subscribe domain.

JMS 2.0 **IBM MQ resource adapter for JMS 2.0**

The Java Platform, Enterprise Edition Connector Architecture (JCA) provides a standard way of connecting applications that are running in a Java EE environment to an Enterprise Information System (EIS) such as IBM MQ or Db2. The IBM MQ resource adapter for JMS 2.0 implements the JCA 1.7 interfaces and contains the IBM MQ classes for JMS. It allows JMS applications and message driven beans (MDBs), running in an application server, to access the resources of an IBM MQ queue manager. The resource adapter supports both the point-to-point domain and the publish/subscribe domain.

The IBM MQ resource adapter supports two types of communication between an application and a queue manager:

Outbound communication

An application starts a connection to a queue manager, and then sends JMS messages to JMS destinations and receives JMS messages from JMS destinations in a synchronous manner.

Inbound communication

A JMS message that arrives at a JMS destination is delivered to an MDB, which processes the message asynchronously.

The resource adapter also contains the IBM MQ classes for Java. The classes are automatically available to applications that are running in an application server that the resource adapter has been deployed into, and allow applications that are running in that application server to use the IBM MQ classes for Java API when they are accessing resources of an IBM MQ queue manager.

The use of the IBM MQ classes for Java within a Java EE environment is supported with restrictions. For information about these restrictions, see "Running IBM MQ classes for Java applications within Java EE" on page 337.

Which version of the resource adapter to use

The version of the resource adapter that you use depends on whether you are deploying it into an application server that supports Jakarta EE or Java EE:

JMS 3.0 **Jakarta EE**

From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported. The IBM MQ resource adapter for Jakarta Messaging must be deployed within an application server that supports Jakarta EE.

Java EE 7

The IBM MQ 8.0 and later resource adapter supports JCA v1.7 and provides JMS 2.0 support. This resource adapter needs to be deployed within a Java EE 7 and later application server (see "IBM MQ resource adapter statement of support" on page 422).

You can install the IBM MQ 8.0 or later resource adapter on any application server that is certified as compliant with the Java Platform, Enterprise Edition 7 specification. Using the IBM MQ 8.0 or later resource adapter, an application can connect to a queue manager using either the BINDINGS or CLIENT transport.

Important: The IBM MQ 8.0 or later resource adapter can be deployed only into an application server that supports JMS 2.0.

Using the resource adapter with WebSphere Application Server traditional

The IBM MQ resource adapter is pre-installed within WebSphere Application Server traditional 9.0 or later. Therefore, there is no requirement to install a new resource adapter.

JMS 3.0 WebSphere Application Server traditional does not currently support Jakarta EE. See IBM MQ resource adapter statement of support.

Note: An IBM MQ 9.0 or later resource adapter can connect in CLIENT or BINDINGS transport mode to any in-service IBM MQ queue manager.

Using the resource adapter with WebSphere Liberty

To connect to IBM MQ from WebSphere Liberty, you must use the IBM MQ resource adapter. Since Liberty does not contain the IBM MQ resource adapter, you must obtain it separately from Fix Central.

The version of the resource adapter that you use depends on whether you are deploying it into a version of Liberty that supports Jakarta EE or Java EE.

For more information about how to download and install the resource adapter, see "Installing the resource adapter in Liberty" on page 429.

Related concepts

"Configuring the resource adapter for inbound communication" on page 436
To configure inbound communication, define the properties of one or more ActivationSpec objects.

"Configuring the resource adapter for outbound communication" on page 453
To configure outbound communication, define the properties of a ConnectionFactory object and an administered destination object.

"Using IBM MQ classes for JMS/Jakarta Messaging" on page 79
IBM MQ classes for JMS and IBM MQ classes for Jakarta Messaging are the Java messaging providers supplied with IBM MQ. As well as implementing the interfaces defined in the JMS and Jakarta Messaging specifications, these messaging providers add two sets of extensions to the Java messaging API.

"Using IBM MQ classes for Java" on page 335
Use IBM MQ in a Java environment. IBM MQ classes for Java allow a Java application to connect to IBM MQ as an IBM MQ client, or connect directly to an IBM MQ queue manager.

Related reference

Configuring the application server to use the latest resource adapter maintenance level
Problem determination for the IBM MQ resource adapter

WebSphere Application Server topics

Maintaining the IBM MQ resource adapter

Deploying JMS applications to Liberty to use the IBM MQ messaging provider

IBM MQ resource adapter statement of support

The IBM MQ resource adapter that you must use for communication between an application and a queue manager depends on whether you are using the Jakarta Messaging 3.0 API or the JMS 2.0 API.

JMS 2.0 IBM MQ 8.0 or later comes with a resource adapter that implements the JMS 2.0 specification. It can be deployed only into an application server that is Java Platform, Enterprise Edition 7 (Java EE 7) compliant and therefore supports JMS 2.0. A list of certified application servers for Java Platform, Enterprise Edition is maintained on Oracle's web site.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. In addition to the resource adapter that supports Java EE and JMS 2.0, IBM MQ 9.3.0 and later provide a resource adapter that supports Jakarta Messaging. It is not supported to use both the Jakarta Messaging 3.0 API and the JMS 2.0 API in the same application. For more information, see Using IBM MQ classes for JMS.

Deployment within WebSphere Liberty

WebSphere Liberty 8.5.5 Fix Pack 6 and later, and WebSphere Application Server Liberty 9.0 and later are Java EE 7 certified application servers so the IBM MQ 9.0 resource adapter can be deployed into them.

To use the IBM MQ resource adapter for Jakarta Messaging with Liberty, you must use a version of Liberty that supports Jakarta EE.

WebSphere Liberty has the following features available for working with resource adapters:

- JMS 3.0 The messaging-3.0 feature to allow working with Jakarta Messaging 3.0 resource adapters.
- The wmqJmsClient-2.0 feature to allow working with JMS 2.0 resource adapters.
- The wmqJmsClient-1.1 feature to allow working with JMS 1.1 resource adapters.

Important:

- JMS 3.0 The IBM MQ resource adapter for Jakarta Messaging must be deployed into a version of Liberty that supports Jakarta EE. This resource adapter cannot be used with versions of Liberty that support the older Java EE specification not Jakarta EE.
- JMS 2.0 The IBM MQ 8.0 or later resource adapter that supports JMS 2.0 must be deployed with the wmqJmsClient-2.0 feature.

Deployment within WebSphere Application Server traditional

WebSphere Application Server traditional 9.0 is supplied with an IBM MQ 9.0 resource adapter already installed. Therefore, there is no requirement to install a new resource adapter. The installed resource adapter can connect in CLIENT or BINDINGS transport mode to any queue managers that are running on a supported version of IBM MQ. For more information, see “Connectivity to IBM MQ 8.0 or later queue managers” on page 423.

Important:

- The IBM MQ 9.0 resource adapter cannot be deployed into versions of WebSphere Application Server traditional before IBM MQ 9.0, because these versions are not Java EE 7 certified.
- JMS 3.0 WebSphere Application Server traditional does not currently support Jakarta EE.

For more information about the versions of the resource adapter that are shipped with WebSphere Application Server, see the technote Which version of WebSphere MQ Resource Adapter (RA) is shipped with WebSphere Application Server?

Using the resource adapter with other application servers

For all other Java EE 7 or Jakarta EE compliant application servers, problems that occur following the successful completion of the IBM MQ resource adapter Installation Verification Test (IVT) can be reported to IBM for the investigation of IBM MQ product trace and other IBM MQ diagnostic information. If the IBM MQ resource adapter IVT cannot be run successfully, any problems that are encountered are likely to be caused by incorrect deployment or incorrect resource definitions that are application server specific and the problems must be investigated by using the application server documentation and the support organization for that application server.

Java Runtime

The Java Runtime (JRE) that is used to run the application server must be one that is supported with the IBM MQ 9.0 or later client. For more information, see System Requirements for IBM MQ. (Select which version and operating system or component report you want to see then follow the **Java** link that is listed under the **Supported Software** tab.)

Connectivity to IBM MQ 8.0 or later queue managers

The full range of JMS 2.0 functionality is available when connecting to an IBM MQ 8.0 or later queue manager by using the resource adapter that has been deployed into a Java EE 7 certified application server. To make use of the JMS 2.0 functionality, the resource adapter needs to connect to the queue manager by using IBM MQ messaging provider normal mode. For more information, see Configuring the JMS **PROVIDERVERSION** property.

JMS 3.0 The full range of Jakarta Messaging 3.0 functionality is available when connecting to an IBM MQ 9.3 or later queue manager by using the resource adapter that has been deployed into a Jakarta EE certified application server.

MQ Extensions

The JMS 2.0 specification introduces changes to how certain behaviors work. Because IBM MQ 8.0 or later implements this specification, there are changes in behavior between IBM MQ 8.0 and later, and earlier versions of the product. In IBM MQ 8.0 or later, the IBM MQ classes for JMS include support for the Java system property `com.ibm.mq.jms.SupportMQExtensions` that, when set to TRUE, causes these versions of IBM MQ to revert these behaviors to those of IBM WebSphere MQ 7.5 or earlier. The default value of the property is FALSE.

The IBM MQ 9.0 or later resource adapter also includes a resource adapter property called `supportMQExtensions` that has the same effect and default value as the `com.ibm.mq.jms.SupportMQExtensions` Java system property. This resource adapter property is set to false in the `ra.xml` by default.

If both the resource adapter property and Java system property are set, then the system property has precedence.

Note that within the resource adapter that is already deployed within WebSphere Application Server traditional 9.0, this property is automatically set to TRUE to aid migration.

For more information, see “SupportMQExtensions property” on page 316.

General issues

Session interleaving is not supported

Some application servers provide a capability called session interleaving, where the same JMS session can be used in multiple transactions, although it is only enlisted in one at a time. The IBM MQ resource adapter does not support this capability, which can lead to the following issues:

```
An attempt to put a message to a MQ queue fails with reason code 2072
(MQRC_SYNCPOINT_NOT_AVAILABLE).
```

Calls to xa_close() fail with reason code -3 (XAER_PROTO), and an FDC with probe ID AT040010 is generated on the IBM MQ queue manager being accessed from the application server. For information on how to disable this capability, see your application server documentation.

Java Transaction API (JTA) specification of how XA resources are recovered for XA transaction recovery

Section 3.4.8 of the JTA specification does not define a specific mechanism by which XA resources are re-created to perform XA transactional recovery. As such, it is up to each individual transaction manager (and, therefore, the application server) how XA resources involved in an XA transaction are recovered. It is possible that, for some application servers, the IBM MQ 9.0 resource adapter does not implement the application server specific mechanisms that are used to perform XA transactional recovery.

Matching connections in a ManagedConnectionFactory

An application server can invoke the matchManagedConnections method on a ManagedConnectionFactory instance provided by the IBM MQ resource adapter. A ManagedConnection is returned only if the method finds one that matches both the **`javax.security.auth.Subject`** and **`javax.resource.spi.ConnectionRequestInfo`** arguments that were passed to the method by the application server.

Limitations of the IBM MQ resource adapter

The IBM MQ resource adapter is supported on all IBM MQ platforms. However, when you use the IBM MQ resource adapter, some features of IBM MQ are unavailable or limited.

The IBM MQ resource adapter has the following limitations:

- Since IBM MQ 8.0, the resource adapter is a Java Platform, Enterprise Edition 7 (Java EE 7) resource adapter providing JMS 2.0 function. Consequently, the IBM MQ 8.0 or later resource adapter must be installed in a Java EE 7 or later certified application server. It can connect in client or bindings transport mode to any in-service queue manager.
- When running inside the WebSphere Liberty application server, the stabilized IBM MQ classes for Java are not supported. Within other application servers the IBM MQ classes for Java are not recommended for use. See the IBM technote Using WebSphere MQ Java Interfaces in J2EE/JEE Environments for details of IBM MQ classes for Java considerations within Java EE.
- When running inside the WebSphere Liberty application server on z/OS, the wmqJmsClient-2.0 feature must be used. Generic JCA support is not possible for z/OS.
- The IBM MQ resource adapter does not support channel exit programs that are written in languages other than Java.
- While an application server is running, the value of the sslFipsRequired property must be true for all JCA resources or false for all JCA resources. This is a requirement even if the JCA resources are not used concurrently. If the sslFipsRequired property has different values for different JCA resources, IBM MQ issues the reason code MQRC_UNSUPPORTED_CIPHER_SUITE, even if a TLS connection is not being used.
- You cannot specify more than one keystore for an application server. If connections are made to more than one queue manager, all the connections must use the same keystore. This limitation does not apply to WebSphere Application Server.
- If you use a client channel definition table (CCDT) with more than one suitable client connection channel definition, in the event of a failure the resource adapter might select a different channel definition and therefore a different queue manager from the CCDT, which would cause problems for

transaction recovery. The resource adapter does not take any action to prevent such a configuration from being used, and it is your responsibility to avoid configurations that might cause problems for transaction recovery.

- The connection retry functionality is not supported for outbound connections when running in a Java EE container (EJB/Servlet). Connection retry is not supported at all for outbound JMS when the adapter is used in a JEE container context, regardless of transaction configuration or for non-transacted use.
- Re-authentication, as defined in Section 9.1.9 of the Java EE Connector Architecture version 1.7 specification, of JMS connections is not supported. The `ra.xml` file within the IBM MQ resource adpater must have the property called **`reauthentication-support`** set to the value `false`. An attempt by the application server to re-authenticate a JMS connection results in the IBM MQ resource adapter throwing a javax.resource.spi.SecurityException with the MQJCA1028 message code.

Related tasks

Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client

Related reference

Federal Information Processing Standards (FIPS) for AIX, Linux, and Windows

WebSphere Application Server and the IBM MQ resource adapter

The IBM MQ resource adapter is used by applications that perform JMS messaging with the IBM MQ messaging provider in WebSphere Application Server.

Important: Do not use the IBM MQ resource adapter with WebSphere Application Server 6.0 or WebSphere Application Server 6.1.

WebSphere Application Server traditional 9.0 includes a version of the IBM MQ 9.0 resource adapter. The IBM MQ 9.0 or later resource adapter cannot be deployed into earlier versions of WebSphere Application Server, as these versions are not Java EE 7 certified.

JMS 3.0 WebSphere Application Server traditional does not currently support Jakarta EE. See IBM MQ resource adapter statement of support.

If you want to use a JMS application to access the resources of an IBM MQ queue manager from within WebSphere Application Server, use the IBM MQ messaging provider in WebSphere Application Server. The IBM MQ messaging provider contains a version of the IBM MQ classes for JMS. For more information, see the technote Which version of WebSphere MQ Resource Adapter (RA) is shipped with WebSphere Application Server?.

Important: Do not include any of the IBM MQ classes for JMS or IBM MQ classes for Java JAR files within your application. Doing so can result in ClassCastExceptions and can be difficult to maintain.

Liberty and the IBM MQ resource adapter

The IBM MQ resource adapter can be installed into WebSphere Liberty by using a Liberty feature. The feature that you use depends on which version of the resource adapter you are installing. Alternatively you can, subject to some restrictions, install the resource adapter by using generic Java Platform, Enterprise Edition Connector Architecture (Java EE JCA) support.

General restrictions when installing the resource adapter into Liberty

The following restrictions apply to the resource adapter when using either the wmqJmsClient-1.1 or wmqJmsClient-2.0 feature and also when using generic JCA support:

- The IBM MQ classes for Java are not supported in Liberty. They must not be used with either the IBM MQ Liberty messaging feature or with the generic JCA support. For more information, see Using WebSphere MQ Java Interfaces in J2EE/JEE Environments.
- The IBM MQ resource adapter has a transport type of BINDINGS_THEN_CLIENT. This transport type is not supported within the IBM MQ Liberty messaging feature.

- Before IBM MQ 9.0, the Advanced Message Security (AMS) feature was not included in the IBM MQ Liberty messaging feature. However, AMS is supported with an IBM MQ 9.0 or later resource adapter.

Note: On IBM MQ versions greater than IBM MQ 9.0.0.6 and IBM MQ 9.1.0.1 you should use the transportSecurity-1.0 feature instead of the ssl-1.0 feature.

For more information, see:

Enabling SSL communication in Liberty
SSL defaults in Liberty
Transport Security 1.0

Restrictions when using the Liberty features

With WebSphere Liberty 8.5.5 Fix Pack 2 to WebSphere Liberty 8.5.5 Fix Pack 5 inclusive, only the wmqJmsClient-1.1 feature was available and only JMS 1.1 could be used. WebSphere Liberty 8.5.5 Fix Pack 6 added the wmqJmsClient-2.0 feature so JMS 2.0 could be used.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported. To use the IBM MQ resource adapter for Jakarta Messaging with Liberty, you must use a version of Liberty that supports Jakarta EE. You must use the resource adapter for Jakarta Messaging with the Liberty generic messaging-3.0 feature.

The feature that you must use depends on which version of the resource adapter you are using:

- The IBM MQ 8.0.0 Fix Pack 3 and later IBM MQ 8.0 resource adapter can be used with wmqJmsClient-2.0 feature only.
- The IBM MQ 9.0 resource adapter can be used with wmqJmsClient-2.0 feature only.
- JMS 3.0 The messaging-3.0 feature allows working with Jakarta Messaging 3.0 resource adapters.

Restrictions when using generic JCA support

If you are using generic JCA support, the following restrictions apply:

- You must specify the level of JMS when using the generic JCA support. JMS 2.0 and JCA 1.7 must be used only with the IBM MQ 8.0.0 Fix Pack 3 and later IBM MQ 8.0 resource adapters.
- It is not possible to run the IBM MQ resource adapter on z/OS using generic JCA support. In order to run the IBM MQ resource adapter on z/OS, it must be run with the wmqJmsClient-1.1 or wmqJmsClient-2.0 feature.
- The location of the resource adapter is specified by using the following xml element:

JMS 3.0
```
<resourceAdapter id="mqJms" location="${server.config.dir}/
wmq.jakarta.jmsra.rar">
  <classloader apiTypeVisibility="spec, ibm-api, api, third-party"/>
</resourceAdapter>
```

JMS 2.0
```
<resourceAdapter id="mqJms" location="${server.config.dir}/wmq.jmsra.rar">
  <classloader apiTypeVisibility="spec, ibm-api, api, third-party"/>
</resourceAdapter>
```

Important: The value of the ID tag can be anything EXCEPT for `wmqJms`. If you do use `wmqJms` as the ID, then Liberty is not able to properly load the resource adapter. This is because `wmqJms` is the ID that is used internally to refer to the specific feature for IBM MQ. It actually creates a NullPointerException.

The following examples show some snippets from a `server.xml` file when running JMS 2.0:

```
<!-- Enable features -->
  <featureManager>
    <feature>servlet-3.1</feature>
    <feature>jndi-1.0</feature>
    <feature>jca-1.7</feature>
```

```
    <feature>jms-2.0</feature>
  </featureManager>
```

Tip: Note the use of the jca-1.7 and jms-2.0 features and the lack of the wmqJmsClient-2.0 feature.

```
<resourceAdapter id="mqJms" location="${server.config.dir}/wmq.jmsra.rar">
  <classloader apiTypeVisibility="spec, ibm-api, api, third-party"/>
</resourceAdapter>
```

Tip: Note the use of `mqJms` for the ID, which is preferred. Do not use `wmqJms`.

```
<application id="WMQHTTP" location="${server.config.dir}/apps/WMQHTTP.war"
name="WMQHTTP" type="war">
    <classloader apiTypeVisibility="spec, ibm-api, api, third-party"
classProviderRef="mqJms"/>
</application>
```

Tip: Note the classloaderProviderRef back to the resource adapter through the ID `mqJms`; this is to permit IBM MQ-specific classes to be loaded.

Restrictions when tracing using generic JCA support

Tracing, and logging are not integrated within the Liberty trace system. Instead, the IBM MQ resource adapter trace must be enabled by using either Java system properties, or an IBM MQ classes for JMS configuration file, as described in Tracing IBM MQ classes for JMS applications. For details about how to set Java system properties in Liberty, see the WebSphere Liberty documentation.

For example, in order to enable trace of IBM MQ resource adapter in Liberty 19.0.0.9, add an entry to the Liberty file `jvm.options`:

1. Create a text file named `jvm.options`.
2. Insert the following JVM options to enable tracing, one per line, into this file:

   ```
   -Dcom.ibm.msg.client.commonservices.trace.status=ON
   -Dcom.ibm.msg.client.commonservices.trace.outputName=C:\Trace\MQRA-WLP_%PID%.trc
   ```

3. To apply these settings to a single server, save `jvm.options` at:

   ```
   ${server.config.dir}/jvm.options
   ```

   To apply these changes to all Liberty, save `jvm.options` at:

   ```
   ${wlp.install.dir}/etc/jvm.options
   ```

   This will take effect for all JVMs that do not have a locally defined `jvm.options` file.
4. Restart the server to enable changes.

This results in trace being written to a trace file called `MQRA-WLP_<process identifier>.trc` in the directory `<path_to_trace_to>`.

Full Liberty XA support with client channel definition tables

When using WebSphere Liberty 18.0.0.2 or later, you can make use of queue manager groups within the client channel definition table (CCDT) in conjunction with XA transactions. This means that it is now possible to make use of workload distribution and availability, provided by queue manager groups, whilst maintaining transaction integrity.

In the event of connectivity errors to a queue manager, the queue manager needs to become available again so that the transaction can be resolved. The transaction recovery is managed by Liberty, and you might need to configure the transaction manager, so that an appropriate period of time is allowed for the queue managers to become available again. For more information, see Transaction manager (transaction) in the WebSphere Liberty product documentation.

This is a client-side feature, that is, you need a resource adapter, not a queue manager.

Installing the IBM MQ resource adapter

The IBM MQ resource adapter is supplied as a resource archive (RAR) file. Install the RAR file in your application server. You might need to add directories to the system path.

About this task

The IBM MQ resource adapter is supplied as a resource archive (RAR) file:

- JMS 3.0 For Jakarta Messaging 3.0, this file is called `wmq.jakarta.jmsra.rar`. The RAR file contains IBM MQ classes for Jakarta Messaging and the IBM MQ implementation of the Jakarta Connectors Architecture (JCA) interfaces.
- JMS 2.0 For JMS 2.0, this file is called `wmq.jmsra.rar`. The RAR file contains IBM MQ classes for JMS and the IBM MQ implementation of the Java EE Connector Architecture (JCA) interfaces.

When you install the resource adapter as part of the IBM MQ product installation, the RAR file is installed with IBM MQ classes for JMS in the directory shown in Table 61 on page 428.

Table 61. IBM MQ directory containing the RAR file for each platform

Platform	Directory
AIX and Linux	*MQ_INSTALLATION_PATH*/java/lib/jca
IBM i	/QIBM/ProdData/mqm/java/lib/jca
Windows	*MQ_INSTALLATION_PATH*\java\lib\jca
z/OS	*MQ_INSTALLATION_PATH*/java/lib/jca

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

You must use the IBM MQ resource adapter to connect to IBM MQ from an application server. Depending on which application server you are using, the resource adapter might be pre-installed, or you might need to install it yourself.

Table 62. Installation of the resource adapter in an application server

Application server	Pre-installed or need to install?
WebSphere Application Server traditional 9.0	The IBM MQ 9.0 resource adapter is pre-installed within WebSphere Application Server traditional 9.0. Therefore, you do not need to install a new resource adapter in WebSphere Application Server traditional 9.0.
WebSphere Liberty	WebSphere Liberty does not contain the IBM MQ resource adapter, so you must obtain it separately from Fix Central.
Other Java EE or Jakarta EE application server	Obtain the resource adapter separately from Fix Central, as for WebSphere Liberty.

Procedure

- If you are connecting to IBM MQ from WebSphere Liberty, or another Java EE or Jakarta EE application server, download and install the IBM MQ resource adapter as described in "Installing the resource adapter in Liberty" on page 429.
- Linux AIX
  For bindings connections on AIX and Linux systems, ensure that the directory containing the Java Native Interface (JNI) libraries is in the system path.

For the location of this directory, which also contains the IBM MQ classes for JMS libraries, see “Configuring the Java Native Interface (JNI) libraries” on page 92.

Windows On Windows, this directory is automatically added to the system path during the installation of IBM MQ classes for JMS.

Tip: As an alternative to setting the system path, the IBM MQ resource adapter has a property called nativeLibraryPath that can be used to specify the location of the JNI library. For example in WebSphere Liberty this would be configured as shown in the following example:

```
<wmqJmsClient nativeLibraryPath="/opt/mqm/java/lib64"/>
```

Transactions are supported in both client and bindings mode.

Installing the resource adapter in Liberty

To connect to IBM MQ from WebSphere Liberty, or other Java EE or Jakarta EE application servers, you must use the IBM MQ resource adapter. Since Liberty does not contain the IBM MQ resource adapter, you must obtain it separately from Fix Central.

Before you begin

Note: The information in this topic does not apply to WebSphere Application Server traditional 9.0. The IBM MQ 9.0 resource adapter is preinstalled within WebSphere Application Server traditional 9.0. Therefore, there is no requirement to install a new resource adapter in this case.

Before you start this task, make sure that you have a Java runtime environment (JRE) installed on your machine and that the JRE has been added to the system path.

The Java installer that is used in this installation process does not require running as root or any specific user. The only requirement is that the user it is run as has access write to the directory that you want the files to go in.

For Liberty versions up to WebSphere Liberty 8.5.5 Fix Pack 1, if an EJB is deployed using solely the configuration within the `ejb-jar.xml`, the version of WebSphere Application Server that the Liberty Profile is using must have APAR PM89890 applied. This method of configuration is used for the resource adapter's installation verification program (IVT), so this APAR is required in order for the IVT to run.

JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported. To use the IBM MQ resource adapter for Jakarta Messaging with Liberty, you must use a version of Liberty that supports Jakarta EE. For example, you can use the Liberty generic messaging-3.0 feature.

About this task

The JAR file for the resource adapter that you can download from Fix Central is executable. When you run this executable file, it displays the IBM MQ license agreement, which must be accepted. It asks for a directory in which to install the IBM MQ resource adapter. The resource adapter RAR file and installation verification test (IVT) program are then installed in that directory. You can either accept the default or specify another directory, which might be the resource adapters directory of an application server, or any other directory on your system. The directory is created as part of the installation if it does not exist.

Before IBM MQ 9.0, the name of the file to be downloaded was in the format of *V.R.M.F*`-WS-MQ-Java-InstallRA.jar`, for example `8.0.0.6-WS-MQ-Java-InstallRA.jar`. From IBM MQ 9.0, the format of the file name is *V.R.M.F*`-IBM-MQ-Java-InstallRA.jar`, for example `9.0.0.0-IBM-MQ-Java-InstallRA.jar`.

After you have downloaded and installed the resource adapter, you are ready to configure it in WebSphere Liberty.

Procedure

1. Download the IBM MQ resource adapter from Fix Central.
   a) Click this link: IBM MQ Resource Adapter.
   b) Find the resource adapter for your version of IBM MQ in the displayed list of available fixes.
      For example:

      ```
      release level: 9.1.4.0-IBM-MQ-Java-InstallRA
      Continuous Delivery Release: 9.1.4 IBM MQ Resource Adapter for use with Application
      Servers
      ```

      Then click the resource adapter file name and follow the download process.
2. Start the installation by entering the following command from the directory to which you downloaded the file.
   From IBM MQ 9.0, the format of the command is as follows:

   ```
   java -jar V.R.M.F-IBM-MQ-Java-InstallRA.jar
   ```

   where *V.R.M.F* is the Version, Release, Modification, and Fix Pack number and *V.R.M.F*`-IBM-MQ-Java-InstallRA.jar` is the name of the file that was downloaded from Fix Central.
   For example, to install the IBM MQ resource adapter for the IBM MQ 9.1.4 release, you would use the following command:

   ```
   java -jar 9.1.4.0-IBM-MQ-Java-InstallRA.jar
   ```

   Note: To carry out this installation, you must have a JRE installed on your machine and added to the system path.

   When you enter the command, the following information is displayed:

   ```
   Before you can use, extract, or install IBM MQ 9.1, you must accept
   the terms of 1. IBM International License Agreement for Evaluation of
   Programs 2. IBM International Program License Agreement and additional
   license information. Please read the following license agreements carefully.

   The license agreement is separately viewable using the
   --viewLicenseAgreement option.
   Press Enter to display the license terms now, or 'x' to skip.
   ```
3. Review and accept the license terms:
   a) To display the license, press Enter.
      Alternatively, pressing x skips the display of the license.
      After display of the license or immediately after selecting x, the following message appears to tell you that you can choose to display additional license terms:

      ```
      Additional license information is separately viewable using the
      --viewLicenseInfo option.
      Press Enter to display additional license information now, or 'x' to skip.
      ```
   b) To display the additional license terms, press Enter.
      Alternatively, pressing x skips the display of the additional license terms.
      After display of the additional license terms or immediately after selecting x, the following message is displayed asking you to accept the license agreement:

      ```
      By choosing the "I Agree" option below, you agree to the terms of the
      license agreement and non-IBM terms, if applicable. If you do not
      agree, select "I do not Agree".

      Select [1] I Agree, or [2] I do not Agree:
      ```
   c) To accept the license agreement and continue with selecting the installation directory, select 1.
      Alternatively, if you select 2 the installation terminates immediately.

If you selected 1, the following message appears, asking you to select a target installation directory:

```
Enter directory for product files or leave blank to accept the default value.
The default target directory is H:\Liberty\WMQ
Target directory for product files?
```

4. Specify the installation directory for the resource adapter:

   - If you want to install the resource adapter in the default location, press Enter without specifying a value.
   - If you want to install the resource adapter in a different location from the default, specify the name of the directory in which you want to install the resource adapter and then press Enter.

   After the files have been installed in the selected location, a confirmation message is displayed as shown in the following example:

   ```
   Extracting files to H:\Liberty\WMQ\wmq
   Successfully extracted all product files.
   ```

   During the installation, a new directory with the name `wmq` is created within the selected installation directory, and the following files are then installed in the `wmq` directory:

   - The installation verification test program, `wmq.jakarta.jmsra.ivt` (Jakarta Messaging 3.0) or `wmq.jmsra.ivt` (JMS 2.0).
   - The IBM MQ RAR file,`wmq.jakarta.jmsra.rar` (Jakarta Messaging 3.0 or `wmq.jmsra.rar` (JMS 2.0).

5. JMS 2.0

   Optional: Configure the Java EE 7 (JMS 2.0) resource adapter in WebSphere Liberty Profile.

   The steps that you must take to configure the resource adapter in Liberty are as follows. For more information, see the WebSphere Application Server product documentation.

   a) Add the wmqJmsClient-2.0 feature to the `server.xml` file to allow working with the IBM MQ resource adapter.

      For more information, see "Which version of the resource adapter to use" on page 421.

   b) Add a reference to the `wmq.jmsra.rar` (JMS 2.0) file that you have installed.

   An example configuration to support servlets and MDBs, with JNDI might look like this:

   ```
    <featureManager>
       <feature>wmqJmsClient-2.0</feature>
       <feature>servlet-3.0</feature>
       <feature>jmsMdb-3.1</feature>
       <feature>jndi-1.0</feature>
       </featureManager>

   <variable name="wmqJmsClient.rar.location"
         value="H:\Liberty\WMQ\wmq\wmq.jmsra.rar"/>
   ```

6. JMS 3.0

   Optional: Configure the Jakarta EE 9 (Jakarta Messaging 3.0) resource adapter in WebSphere Liberty Profile.

   The steps that you must take to configure the resource adapter in Liberty are as follows. For more information, see the WebSphere Application Server product documentation.

   a) Add the wmqJmsClient-3.0 feature to the `server.xml` file to allow working with the IBM MQ resource adapter.

      For more information, see "Which version of the resource adapter to use" on page 421.

   b) Add a reference to the `wmq.jakarta.jmsra.rar` (Jakarta Messaging 3.0) file that you have installed.

   An example configuration to support servlets and MDBs, with JNDI might look like this:

   ```
    <featureManager>
       <feature>wmqJmsClient-3.0</feature>
   ```

```
    <feature>servlet-3.0</feature>
    <feature>jmsMdb-3.1</feature>
    <feature>jndi-1.0</feature>
    </featureManager>

<variable name="wmqJmsClient.rar.location"
      value="H:\Liberty\WMQ\wmq\wmq.jmsra.rar"/>
```

Note: If you are using Open Liberty, rather than WebSphere Liberty Profile, you will need to use the generic resource adapter support feature "messagingClient-3.0" in place of "wmqJmsClient-3.0" and other aspects of the configuration will be different. Please refer to the Open Liberty documentation for more details.

Configuring the IBM MQ resource adapter

To configure the IBM MQ resource adapter, you define various Java Platform, Enterprise Edition Connector Architecture (JCA) resources and, optionally, system properties. You must also configure the resource adapter to run the installation verification test (IVT) program. This is important because IBM service might require this program to be run to indicate that any non-IBM application server has been correctly configured.

Before you begin

This task assumes that you are already familiar with JMS and IBM MQ classes for JMS. Many of the properties used to configure the IBM MQ resource adapter are equivalent to properties of IBM MQ classes for JMS objects and have the same function.

About this task

Every application server provides its own set of administration interfaces. Some application servers provide graphical user interfaces to define JCA resources, but others require the administrator to write XML deployment plans. It is therefore beyond the scope of this documentation to provide information about how to configure the IBM MQ resource adapter for each application server.

The following steps therefore focus only on what you need to configure. Refer to documentation supplied with your application server for information about how to configure a JCA resource adapter.

Procedure

Define JCA resources in the following categories:

- Define the properties of the ResourceAdapter object.

  These properties, which represent the global properties of the resource adapter, such as the level of diagnostic tracing, are described in “Configuration for ResourceAdapter object properties” on page 433.
- Define the properties of an ActivationSpec object.

  These properties determine how an MDB is activated for inbound communication. For more information, see “Configuring the resource adapter for inbound communication” on page 436.
- Define the properties of a ConnectionFactory object.

  The application server uses these properties to create a JMS ConnectionFactory object for outbound communication. For more information, see “Configuring the resource adapter for outbound communication” on page 453.
- Define the properties of an administered destination object.

  The application server uses these properties to create a JMS Queue object or JMS Topic object for outbound communication. For more information, see “Configuring the resource adapter for outbound communication” on page 453.
- Optional: Define a deployment plan for the resource adapter.

  The IBM MQ resource adapter RAR file contains a file called `META-INF/ra.xml`, which contains a deployment descriptor for the resource adapter. This deployment descriptor is defined by the XML

schema at https://xmlns.jcp.org/xml/ns/javaee/connector_1_7.xsd and contains information about the resource adapter and the services that it provides. An application server might also require a deployment plan for the resource adapter. This deployment plan is specific to the application server.

Specify JVM system properties as required:

- If you are using Transport Layer Security (TLS), specify the locations of the keystore file and truststore file as JVM system properties, as in the following example:

```
java ... -Djavax.net.ssl.keyStore=
key_store_location
         -Djavax.net.ssl.trustStore=trust_store_location
         -Djavax.net.ssl.keyStorePassword=key_store_password
```

  These properties cannot be properties of an ActivationSpec or ConnectionFactory object, and you cannot specify more than one keystore for an application server. The properties apply to the whole JVM, and might therefore affect the application server if other applications, running in the application server, are using TLS connections. The application server might also reset these properties to different values. For more information about using TLS with IBM MQ classes for JMS, see "Using TLS with IBM MQ classes for JMS" on page 246.

- Optional: If required, configure the resource adapter to log warning messages to your application server's standard output log.

  The resource adapter logs, warning, and error messages use the same mechanism as the IBM MQ classes for JMS. For more information, see Logging errors for IBM MQ classes for JMS. This means that, by default, the messages go to a file called `mqjms.log`. To configure the resource adapter to additionally log warning messages to your application server's standard output log, set the following JVM system property for your application server:

```
-Dcom.ibm.msg.client.commonservices.log.outputName=mqjms.log,stdout
```

  This is the same property as the one that is used to control trace for the IBM MQ classes for JMS. As with IBM MQ classes for JMS, it is possible to use a system property pointing to the `jms.config` file (see "The IBM MQ classes for JMS/Jakarta Messaging configuration file" on page 94). For information on how to set a JVM system property, see your application server documentation.

Configure the resource adapter to run the installation verification test

- Configure the resource adapter to run the installation verification test (IVT) program supplied with the IBM MQ resource adapter.

  For information about what you need to configure in order to run the IVT program, see "Verifying the resource adapter installation" on page 473.

  This is important because IBM service might require this program to be run to indicate that any non-IBM application server has been correctly configured.

  Important: You must configure the resource adapter before you can run the program.

Configuration for ResourceAdapter object properties

The ResourceAdapter object encapsulates the global properties of the IBM MQ resource adapter, such as the level of diagnostic tracing. To define these properties, use the facilities of your resource adapter, as described in the documentation supplied with your application server.

The ResourceAdapter object has two sets of properties:

- Properties associated with diagnostic tracing
- Properties associated with the connection pool managed by the resource adapter

The way in which you define these properties depends on the administration interfaces that your application server provides. If you are using WebSphere Application Server traditional, see "WebSphere Application Server traditional configuration" on page 435 or if you are using WebSphere Liberty, see "WebSphere Liberty configuration" on page 435. For other application servers, see the product documentation for your application server.

For more information about defining properties associated with diagnostic trace, see Tracing the IBM MQ Resource Adapter

The resource adapter manages an internal connection pool of JMS connections that are used to deliver messages to MDBs. Table 63 on page 434 lists the properties of the ResourceAdapter object that are associated with the connection pool.

Table 63. Properties of the ResourceAdapter object that are associated with the connection pool

Name of property	Type	Default value	Description
maxConnections	String	50	The maximum number of connections to an IBM MQ queue manager and the maximum number of MDBs deployed.
connectionConcurrency	String	1	The maximum number of MDBs to share a JMS connection. Sharing connections is not possible and this property always has the value 1.
reconnectionRetryCount	String	5	The maximum number of attempts made by the resource adapter to reconnect to an IBM MQ queue manager if a connection fails.
reconnectionRetryInterval	String	300 000	The time, in milliseconds, that the resource adapter waits before trying to reconnect to an IBM MQ queue manager.
startupRetryCount	String	0	The default number of times to try and connect a MDB on startup, if the queue manager is not running when the application server is started.
startupRetryInterval	String	30 000	The default sleep time between startup connection attempts (in milliseconds).
supportMQExtensions	String	false	Reverts the IBM MQ JMS behavior to pre-JMS 2.0 behavior. For more information, see "SupportMQExtensions property" on page 316.
nativeLibraryPath	String	<empty>	The path to use to load the IBM MQ JNI library to permit bindings mode connections. Windows On Windows the system path also needs to contain the location of the matching IBM MQ installation.

When an MDB is deployed in the application server, a new JMS connection is created and a conversation started with the queue manager, provided the maximum number of connections specified by the maxConnection property is not exceeded. The maximum number of MDBs therefore equals the maximum number of connections. If the number of deployed MDBs reaches this maximum, any attempt to deploy another MDB fails. If an MDB is stopped, its connection can be used by another MDB.

In general, if many MDBs are to be deployed, you must increase the value of the maxConnections property.

The reconnectionRetryCount and reconnectionRetryInterval properties govern the behavior of the resource adapter when connections to an IBM MQ queue manager fail, because of a network failure for example. When a connection fails, the resource adapter suspends the delivery of messages to all MDBs supplied by that connection for an interval specified by the reconnectionRetryInterval property. The resource adapter then attempts to reconnect to the queue manager. If the attempt fails, the resource adapter makes further attempts to reconnect at intervals specified by the reconnectionRetryInterval

property until the limit imposed by the reconnectionRetryCount property is reached. If all attempts fail, delivery is stopped permanently until the MDBs are restarted manually.

In general, the ResourceAdapter object requires no administration. However, to enable diagnostic tracing on AIX and Linux systems for example, you can set the following properties:

```
traceEnabled:    true
traceLevel:      10
```

These properties have no effect if the resource adapter has not been started, which is the case, for example, when applications using IBM MQ resources are running only in the client container. In this situation, you can set the properties for diagnostic tracing as Java Virtual Machine (JVM) system properties. You can set the properties by using the -D flag on the **java** command, as in the following example:

```
java ... -DtraceEnabled=true -DtraceLevel=6
```

You do not need to define all the properties of the ResourceAdapter object. Any properties left unspecified take their default values. In a managed environment, it is better not to mix the two ways of specifying properties. If you do mix them, the JVM system properties take precedence over the properties of the ResourceAdapter object.

WebSphere Application Server traditional configuration

The same properties are available for the resource adapter in WebSphere Application Server traditional, but they should be set for within the properties panel of the resource adapter (see JMS provider settings in the WebSphere Application Server traditional product documentation. Trace is controlled by the diagnostics section of the WebSphere Application Server traditional configuration. For more information, see Working with Diagnostic Providers in the WebSphere Application Server traditional product documentation.

WebSphere Liberty configuration

The resource adapter is configured using XML elements in the `server.xml` file, as shown in the following example:

JMS 3.0

```
<featureManager>
...
    <feature>messaging-3.0</feature>
...
</featureManager>
    <variable name="wmqJmsClient.rar.location"
        value="F:/_rtc_wmq8005/_build/ship/lib/jca/wmq.jakarta.jmsra.rar"/>
...
        <wmqJmsClient supportMQExtensions="true" logWriterEnabled="true"/>
```

JMS 2.0

```
<featureManager>
...
    <feature>wmqJmsClient-2.0</feature>
...
</featureManager>
    <variable name="wmqJmsClient.rar.location"
        value="F:/_rtc_wmq8005/_build/ship/lib/jca/wmq.jmsra.rar"/>
...
        <wmqJmsClient supportMQExtensions="true" logWriterEnabled="true"/>
```

Trace is enabled by adding this XML element:

```
<logging traceSpecification="JMSApi=all:WAS.j2c=all:"/>
```

Configuring the resource adapter for inbound communication

To configure inbound communication, define the properties of one or more ActivationSpec objects.

The properties of an ActivationSpec object determine how a message driven bean (MDB) receives JMS messages from an IBM MQ queue. The transactional behavior of the MDB is defined in its deployment descriptor.

An ActivationSpec object has two sets of properties:

- Properties that are used to create a JMS connection to an IBM MQ queue manager
- Properties that are used to create a JMS connection consumer that delivers messages asynchronously as they arrive on a specified queue

The way in which you define the properties of an ActivationSpec object depends on the administration interfaces provided by your application server.

Properties used to create a JMS connection to an IBM MQ queue manager

All of the properties in Table 64 on page 436 are optional.

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection

Name of property	Type	Valid values (default value in bold)	Description
applicationName	String	• The invoking class name, if it is available, adjusted to be no longer than 28 characters. If it is not available, the string `WebSphere MQ Client for Java` is used.	The name by which an application is registered with the queue manager. This application name is shown by the **`DISPLAY CONN MQSC/PCF`** command (where the field is called **`APPLTAG`**) or in the IBM MQ Explorer **Application Connections** display (where the field is called **`App name`**).
brokerCCDurSubQueue [1]	String	• **SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives durable subscription messages
brokerCCSubQueue [1]	String	• **SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives nondurable subscription messages
brokerControlQueue [1]	String	• **SYSTEM.BROKER.CONTROL.QUEUE** • A queue name	The name of the broker control queue
brokerQueueManager [1]	String	• **"" (empty string)** • A queue manager name	The name of the queue manager on which the broker is running
brokerSubQueue [1]	String	• **SYSTEM.JMS.ND.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a nondurable message consumer receives messages

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
brokerVersion [1]	String	• **unspecified** - After the broker is migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration, this property is no longer relevant. • **V1** - To use an IBM MQ publish/subscribe broker.This value is the default value if TRANSPORT is set to BIND or CLIENT. • **V2** - To use a broker of IBM Integration Bus in native mode. This value is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP.	The version of the broker being used
ccdtURL	String	• **null** • A uniform resource locator (URL)	A URL that identifies the name and location of the file containing the client channel definition table (CCDT) and specifies how the file can be accessed
CCSID	String	• **819** • A coded character set identifier supported by the Java virtual machine (JVM)	The coded character set identifier for a connection
channel	String	• **SYSTEM.DEF.SVRCONN** • The name of an MQI channel	The name of the MQI channel to use
cleanupInterval [1]	int	• **3 600 000** • A positive integer	The interval, in milliseconds, between background runs of the publish/subscribe cleanup utility
cleanupLevel [1]	String	• **SAFE** • NONE • STRONG • FORCE • NONDUR	The cleanup level for a broker-based subscription store
clientID	String	• **null** • A client identifier	The client identifier for a connection
cloneSupport	String	• **DISABLED** - Only one instance of a durable topic subscriber can run at a time. • ENABLED - Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).	Whether two or more instances of the same durable topic subscriber can run simultaneously

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
connectionFactoryLookup	String	• **null** • The JNDI name for a ConnectionFactory object	If this property is set, the ActivationSpec looks up a JMS ConnectionFactory object with the specified JNDI name in the JNDI namespace of the application server, and then uses the properties of that object to create a JMS connection to an IBM MQ queue manager, with one exception. The only property of the ActivationSpec that will be used when creating the JMS connection is the clientID. For more information, see "ActivationSpec connectionFactoryLookup and destinationLookup properties" on page 449.
connectionNameList	String	• **localhost(1414)** • A string composed of items separated by commas where each item takes the format: `HOSTNAME(PORT)` where *HOSTNAME* is either a DNS name or an IP address.	A list of TCP/IP connection names used for inbound communications. When specified, **connectionNameList** supersedes the **hostname** and **port** properties. This property is used to reconnect to multi-instance queue managers. **connectionNameList** is similar in form to **localAddress**, but must not be confused with it. **localAddress** specifies the characteristics of the local communications, whereas **connectionNameList** specifies how to reach a remote queue manager.
dynamicallyBalanced [4]	Boolean	• **false** • true	Whether this MDB can be requested to receive messages from a different queue manager as part of application balancing in a uniform cluster.

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
failIfQuiesce	Boolean	• **true** • false	Whether calls to certain methods fail if the queue manager is in a quiescing state
headerCompression	String	• **NONE** • SYSTEM - RLE message header compression is performed	A list of the techniques that can be used for compressing header data on a connection
hostName	String	• **localhost** • A host name • An IP address	The host name or IP address of the system on which the queue manager resides. The **hostname** and **port** properties are superseded by the **connectionNameList** property when it is specified.
localAddress	String	• **null** • A string in the format: `[ host_name ][(low_port [, high_port ])]` where *host_name* is a host name or IP address, *low_port* and *high_port* are TCP port numbers, and brackets denote an optional component	For a connection to a queue manager, this property specifies either or both of the following things: • The local network interface to be used • The local port, or range of local ports, to be used **localAddress** is similar in form to **connectionNameList**, but must not be confused with it. **localAddress** specifies the characteristics of the local communications, whereas **connectionNameList** specifies how to reach a remote queue manager.
messageCompression	String	• **NONE** • A list of one or more of the following values separated by blank characters: RLE ZLIBFAST ZLIBHIGH V 9.4.0 LZ4FAST V 9.4.0 LZ4HIGH	A list of the techniques that can be used for compressing message data on a connection

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
messageRetention [1]	Boolean	• **true** - Unwanted messages remain on the input queue • false - Unwanted messages are dealt with according to their disposition options	Whether the connection consumer keeps unwanted messages on the input queue
messageSelection [1]	String	• **CLIENT** • BROKER	Determines whether message selection is done by IBM MQ classes for JMS or by the broker. Message selection by the broker is not supported when brokerVersion has the value 1.
password	String	• **null** • A password	The default password to use when creating a connection to the queue manager
pollingInterval [1]	int	• **5000** • Any positive integer	If each message listener within a session has no suitable message on its queue, this value is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property. This property is relevant only if TRANSPORT has the value BIND or CLIENT.
port	int	• **1414** • A TCP port number	The port on which the queue manager listens. The **hostname** and **port** properties are superseded by the **connectionNameList** property when it is specified.

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
providerVersion	string	• **unspecified** • A string in one of the following formats – V.R.M.F – V.R.M – V.R – V where V, R, M, and F are integer values greater than or equal to zero.	The version, release, modification level and fix pack of the queue manager to which the MDB intends to connect.
queueManager	String	• **"" (empty string)** • A queue manager name	The name of the queue manager to connect to
receiveExit [3]	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the IBM MQ classes for Java interface, MQReceiveExit	Identifies a channel receive exit program, or a sequence of receive exit programs to be run in succession
receiveExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel receive exit programs when they are called

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
rescanInterval [1]	int	• **5000** • Any positive integer	When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, IBM MQ classes for JMS searches the IBM MQ queue for suitable messages in the sequence determined by the **MsgDeliverySequence** attribute of the queue. When IBM MQ classes for JMS finds a suitable message and delivers it to the consumer, IBM MQ classes for JMS resumes the search for the next suitable message from its current position in the queue. IBM MQ classes for JMS continues to search the queue in this way until it reaches the end of the queue, or until the interval of time in milliseconds, as determined by the value of this property, has expired. In each case, IBM MQ classes for JMS returns to the beginning of the queue to continue its search, and a new time interval commences.
securityExit [3]	String	• **null** • The fully qualified name of a class that implements the IBM MQ classes for Java interface, MQSecurityExit	Identifies a channel security exit program
securityExitInit	String	• **null** • A string of user data	The user data that is passed to a channel security exit program when it is called
sendExit [3]	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the IBM MQ classes for Java interface, MQSendExit	Identifies a channel send exit program, or a sequence of send exit programs to be run in succession

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
sendExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel send exit programs when they are called
shareConvAllowed	Boolean	• NO - A client connection cannot share its socket. • **YES** - A client connection can share its socket.	Whether a client connection can share its socket with other top-level JMS connections from the same process to the same queue manager, if the channel definitions match
sparseSubscriptions [1]	Boolean	• **false** - Subscriptions receive frequent matching messages. • true - Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.	Controls the message retrieval policy of a TopicSubscriber object
sslCertStores	String	• **null** • A string of one or more LDAP URLs separated by blanks. Each LDAP URL has the format: `ldap://host_name [: port ]` where *host_name* is a host name or IP address, *port* is a TCP port number, and brackets denote an optional component.	The Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs) for use on a TLS connection
sslCipherSuite	String	• **null** • The name of a CipherSuite	The CipherSuite to use for a TLS connection
sslFipsRequired [2]	Boolean	• **false** • true	Whether a TLS connection must use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS)
sslPeerName	String	• **null** • A template for distinguished names	For a TLS connection, a template that is used to check the distinguished name in the digital certificate provided by the queue manager
sslResetCount	int	• **0** • An integer in the range 0 - 999 999 999	The total number bytes sent and received by a TLS connection before the secret keys used by TLS are renegotiated

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
sslSocketFactory	String	A string representing the fully qualified class name of a class providing an implementation of the javax.net.ssl.SSLSocketFactory interface. Optionally including an argument to be passed to the constructor method, enclosed in parentheses.	Any connections established in the scope of the administered object use sockets obtained from this implementation of the SSLSocketFactory interface.
statusRefreshInterval [1]	int	• **60000** • Any positive integer	The interval, in milliseconds, between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if **subscriptionStore** has the value QUEUE.
subscriptionStore [1]	String	• **BROKER** • MIGRATE • QUEUE	Determines where IBM MQ classes for JMS stores persistent data about active subscriptions

Table 64. Properties of an ActivationSpec object that are used to create a JMS connection (continued)

Name of property	Type	Valid values (default value in bold)	Description
transportType	String	• **CLIENT** • BINDINGS • BINDINGS_THEN_CLIENT	Whether a connection to a queue manager uses client mode or bindings mode. If the value `BINDINGS_THEN_CLIENT` is specified, the resource adapter first tries to make a connection in bindings mode. If this connection attempt fails the resource adapter then tries to make a client mode connection. z/OS If an activation specification that is running on a WebSphere Application Server for z/OS system has been configured to use the `BINDINGS_THEN_CLIENT` transport mode and a previously established connection is broken, then any reconnection attempts by the activation specification first attempt to use the `BINDINGS` transport mode. If the `BINDINGS` transport mode connection attempt is unsuccessful, the activation specification subsequently attempts a `CLIENT` transport mode connection.
username	String	• **null** • A user name	The default user name to use when creating a connection to a queue manager
wildcardFormat	String	• CHAR- Recognizes character wildcards only, as used in broker version 1 • **TOPIC** - Recognizes topic level wildcards only, as used in broker version 2	Which version of wildcard syntax is to be used

Notes:

1. This property can be used with version 70 of IBM MQ classes for JMS.
2. For important information about using the sslFipsRequired property, see “Limitations of the IBM MQ resource adapter” on page 424.
3. For information on how to configure the resource adapter so that it can locate an exit, see “Configuring IBM MQ classes for JMS to use channel exits” on page 271.

4. The dynamicallyBalanced property is not supported in conjunction with XA transaction support. If dynamicallyBalanced is "true", then the MDB must be configured to disable XA transactions.

Properties used to create a JMS connection consumer

Note: The **destination** and **destinationType** must be defined explicitly. All the other properties in Table 65 on page 446 are optional.

Table 65. Properties of an ActivationSpec object that are used to create a JMS connection consumer

Name of property	Type	Valid values (default value in bold)	Description
destination	String	A destination name	The destination from which to receive messages. The **useJNDI** property determines how the value of this property is interpreted.
destinationLookup	String	• **null** • The JNDI name for a Destination object	If this property is set, the ActivationSpec looks up a JMS Destination object with the specified JNDI name in the JNDI namespace of the application server, and then uses the properties of that object to create a JMS connection consumer, in preference to the other properties specified on the ActivationSpec. For more information, see “ActivationSpec connectionFactoryLookup and destinationLookup properties” on page 449.
destinationType	String	• jakarta.jms.Queue (Jakarta Messaging 3.0) • jakarta.jms.Topic (Jakarta Messaging 3.0) • javax.jms.Queue (JMS 2.0) • javax.jms.Topic (JMS 2.0)	The type of destination, a queue, or a topic
maxMessages	int	• **1** • A positive integer	The maximum number of messages that can be assigned to a server session at one time. If the activation spec is delivering messages to an MDB in an XA transaction, a value of 1 is used regardless of the setting of this property.
maxPoolDepth	int	• **10** • A positive integer	The maximum number of server sessions in the server session pool used by the connection consumer
messageSelector	String	• **null** • An SQL92 message selector expression	A message selector expression specifying which messages are to be delivered

Table 65. Properties of an ActivationSpec object that are used to create a JMS connection consumer (continued)

Name of property	Type	Valid values (default value in bold)	Description
nonASFTimeout	int	• **0** • A positive integer	A positive value indicates that non-ASF delivery is used. The value is the time, in milliseconds, that a get request waits for messages that might not have yet arrived (a get with wait call). The default value, 0, indicates that ASF delivery is used. This parameter is valid if: • The application is running on WebSphere Application Server 7.0 or later. • The application is running in WebSphere Liberty using the appropriate level of wmqJmsClient feature. For more information, see "Liberty and the IBM MQ resource adapter" on page 425.
nonASFRollbackEnabled	Boolean	• **false** - The message is consumed even if the MDB fails • true - Failure within the MDB causes the message to roll back to the queue.	Whether message delivery is within an IBM MQ syncpoint if the MDB is non-transacted. Ignored if the MDB is transacted or if **nonASFTimeout** is set to 0.
poolTimeout	int	• **300000** • A positive integer	The time, in milliseconds, that an unused server session is held open in the server session pool before being closed due to inactivity
readAheadAllowed	int	• **DESTINATION** - Determine whether read ahead is allowed by referring to the queue or topic definition. • DISABLED - Read ahead is not allowed. • ENABLED - Read ahead is allowed. • QUEUE - Determine whether read ahead is allowed by referring to the queue definition. • TOPIC - Determine whether read ahead is allowed by referring to the topic definition.	Whether the activation specification browsing thread is allowed to use read ahead to browse multiple messages from the destination into an internal buffer, before handing off to the server sessions for destructive consumption. **Note:** Enabling read ahead might result in an increase of JMSCC0108 messages, or a reduction in performance, or both if the MDB processing rate cannot keep up with the rate of browsing messages from the destination.

Table 65. Properties of an ActivationSpec object that are used to create a JMS connection consumer (continued)

Name of property	Type	Valid values (default value in bold)	Description
readAheadClosePolicy	int	• **ALL** - All messages in the internal read ahead buffer are delivered to the MDB before it stops. • CURRENT - Only the current MDB invocation completes, potentially leaving messages in the internal read ahead buffer, which are then discarded.	What happens to messages in the internal read ahead buffer when the MDB is stopped by the administrator.
receiveCCSID	int	• **0** - Use JVM `Charset.defaultCharset` • 1208 - UTF-8 • A supported coded character set identifier	Destination property that sets the target CCSID for queue manager message conversion. The value is ignored unless **`receiveConversion`** is set to `QMGR`.
receiveConversion	String	• **CLIENT_MSG** • QMGR	Destination property that determines whether data conversion is going to be performed by the queue manager.
sharedSubscription	Boolean	• **False** - The MDB should not open the subscription as a shared subscription. • True - The MDB should open the subscription as a shared subscription (with the rules that JMS 2.0 implies, see the JMS 2.0 specification at Java.net).	Controls how an MDB is driven from a shared subscription. For more information about how to use this property, see "Examples of how to define the sharedSubscription property" on page 452.
startTimeout	int	• **10 000** • A positive integer	The time, in milliseconds, within which delivery of a message to an MDB must start after the work to deliver the message has been scheduled. If this time elapses, the message is rolled back onto the queue.
subscriptionDurability	String	• **NonDurable** - A nondurable subscription is used to deliver messages to an MDB subscribing to the topic. • Durable - A durable subscription is used to deliver messages to an MDB subscribing to the topic.	Whether a durable or nondurable subscription is used to deliver messages to an MDB subscribing to the topic
subscriptionName	String	• **"" (empty string)** • A subscription name	The name of the durable subscription

Table 65. Properties of an ActivationSpec object that are used to create a JMS connection consumer (continued)			
Name of property	**Type**	**Valid values (default value in bold)**	**Description**
useJNDI	Boolean	• **false** - The property called destination is interpreted as the name of an IBM MQ queue or a topic. • true - The property called destination is interpreted as the name of one of the following objects in the JNDI namespace of the application server: – jakarta.jms.Queue (Jakarta Messaging 3.0) – jakarta.jms.Topic (Jakarta Messaging 3.0) – javax.jms.Queue (JMS 2.0) – javax.jms.Topic (JMS 2.0)	Deprecated Determines how the value of the property called destination is interpreted **Note:** This property is deprecated in IBM MQ 9.0. The destinationLookup property should be used instead.

Property conflicts and dependencies

An ActivationSpec object can have conflicting properties. For example, you can specify TLS properties for a connection in bindings mode. In this case, the behavior is determined by the transport type and the messaging domain, which is either point-to-point or publish/subscribe as determined by the **destinationType** property. Any properties that are not applicable to the specified transport type or messaging domain are ignored.

If you define a property that requires other properties to be defined, but you do not define these other properties, the ActivationSpec object throws an InvalidPropertyException exception when its validate() method is called during the deployment of an MDB. The exception is reported to the administrator of the application server in a manner that depends on the application server. For example, if you set the subscriptionDurability property to Durable, indicating that you want use durable subscriptions, you must also define the **subscriptionName** property.

If the properties called **ccdtURL** and **channel** are both defined, an InvalidPropertyException exception is thrown. However, if you define the **ccdtURL** property only, leaving the property called **channel** with its default value of `SYSTEM.DEF.SVRCONN`, no exception is thrown, and the client channel definition table identified by the **ccdtURL** property is used to start a JMS connection.

ActivationSpec connectionFactoryLookup and destinationLookup properties

The JMS 2.0 specification introduced two new ActivationSpec properties. The connectionFactoryLookup and destinationLookup properties can be provided with a JNDI name of an administered object to be used in preference to the other ActivationSpec properties.

For example, if a connection factory is defined in JNDI and the JNDI name of that object is specified in the connectionFactoryLookup property for an activation specification, all the properties of the connection factory that are defined in JNDI are used in preference to the properties in Table 64 on page 436.

If a destination is defined in JNDI and the JNDI name is set in the ActivationSpec's destinationLookup property then the values of that are used in preference to the values in Table 65 on page 446. For more information about how these two properties are used, see “ActivationSpec connectionFactoryLookup and destinationLookup properties” on page 449.

These two properties can be used to specify the JNDI names of ConnectionFactory and Destination objects that are used in preference to the properties of the ActivationSpec as defined in Table 64 on page 436 and Table 65 on page 446.

It is important to note the following points that describe how these properties work in detail.

connectionFactoryLookup
The ConnectionFactory that is looked up from JNDI is used as a source of the properties listed in Table 64 on page 436. The ConnectionFactory object is not used to actually create any JMS connections, only the properties of the object are queried. These properties from the ConnectionFactory override any properties that are defined on the ActivationSpec. There is a single exception to this. If the ActivationSpec has the **ClientID** property set, then the value of this property overrides the value specified in the ConnectionFactory. This is because a common scenario is using a single ConnectionFactory with multiple ActivationSpecs. This simplifies administration. However, the JMS 2.0 specification states that every JMS Connection created from a ConnectionFactory should have a unique **ClientID**. Because of this, ActivationSpecs need to have the ability to override any value set on the ConnectionFactory. If no **ClientID** is set on the ActivationSpec, any value on the connection factory is used.

destinationLookup
A **Destination** and a **UseJndi** property are defined on the ActivationSpec. If the **UseJndi** flag is set to `true`, then the text specified in the destination property is considered to be a JNDI name and a destination object with that JNDI name is looked up from JNDI.

The destinationLookup property behaves in exactly the same way. If it has been set, then a destination object with the JNDI name specified by the property is looked up from JNDI. This property has precedence over the **useJNDI** property.

Deprecated The useJNDI property is deprecated at IBM MQ 9.0 as the **destinationLookup** property is the JMS 2.0 specification or later equivalent of performing the same function.

ActivationSpec properties with no equivalents in IBM MQ classes for JMS

Most of the properties of an ActivationSpec object are equivalent to properties of IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging objects, or to parameters of IBM MQ classes for JMS IBM MQ classes for Jakarta Messaging methods. However, three tuning properties, and one usability property, have no equivalents in IBM MQ classes for JMS or IBM MQ classes for Jakarta Messaging:

startTimeout
The time, in milliseconds, that the work manager of the application server waits for resources to become available after the resource adapter schedules a Work object to deliver a message to an MDB. If this time elapses before delivery of the message starts, the Work object times out, the message is rolled back onto the queue, and the resource adapter can then attempt to deliver the message again. A warning is written to diagnostic trace, if enabled, but does not otherwise affect the process of delivering messages. You might expect this condition to occur only at times when the application server is experiencing a very high load. If the condition occurs regularly, consider increasing the value of this property to give the work manager longer to schedule message delivery.

maxPoolDepth
The maximum number of server sessions in the server session pool used by a connection consumer. When a server session is created, it starts a conversation with a queue manager. The connection consumer uses a server session to deliver a message to an MDB. A larger pool depth allows more messages to be delivered concurrently in high volume situations, but uses more resources of the application server. If many MDBs are to be deployed, consider making the pool depth smaller in order to maintain the load on the application server at a manageable level. Each connection consumer uses its own server session pool, so that this property does not define the total number of server sessions available to all connection consumers.

poolTimeout
The time, in milliseconds, that an unused server session is held open in the server session pool before being closed due to inactivity. A transient increase in the message workload causes additional server

sessions to be created in order to distribute the load but, after the message workload returns to normal, the additional server sessions remain in the pool and are not used.

Every time a server session is used, it is marked with a timestamp. Periodically a scavenger thread checks that each server session has been used within the period specified by this property. If a server session has not been used, it is closed and removed from the server session pool. A server session might not be closed immediately after the specified period has elapsed, this property represents the minimum period of inactivity before removal.

useJNDI
For a description of this property, see Table 65 on page 446.

Deploying an MDB

To deploy an MDB, first define the properties of an ActivationSpec object, specifying the properties that the MDB requires. The following example is a typical set of properties that you might define explicitly:

JMS 3.0

```
channel:           SYSTEM.DEF.SVRCONN
destination:       SYSTEM.DEFAULT.LOCAL.QUEUE
destinationType:   jakarta.jms.Queue
hostName:          192.168.0.42
messageSelector:   color='red'
port:              1414
queueManager:      ExampleQM
transportType:     CLIENT
```

JMS 2.0

```
channel:           SYSTEM.DEF.SVRCONN
destination:       SYSTEM.DEFAULT.LOCAL.QUEUE
destinationType:   javax.jms.Queue
hostName:          192.168.0.42
messageSelector:   color='red'
port:              1414
queueManager:      ExampleQM
transportType:     CLIENT
```

The application server uses the properties to create an ActivationSpec object, which is then associated with an MDB. The properties of the ActivationSpec object determine how messages are delivered to the MDB. Deployment of the MDB fails if the MDB requires distributed transactions but the resource adapter does not support distributed transactions. For information about how to install the resource adapter so that distributed transactions are supported, see “Installing the IBM MQ resource adapter” on page 428.

If more than one MDB is receiving messages from the same destination, then a message sent in the point-to-point domain is received by only one MDB, even if other MDBs are eligible to receive the message. In particular, if two MDBs are using different message selectors, and an incoming message matches both message selectors, only one of the MDBs receives the message. The MDB chosen to receive a message is undefined, and you cannot rely on a specific MDB receiving the message. Messages sent in the publish/subscribe domain are received by all eligible MDBs.

In some circumstances, a message delivered to an MDB might be rolled back onto an IBM MQ queue. This roll back can happen, for example, if a message is delivered within a unit of work that is then rolled back. A message that is rolled back is delivered again, but a badly formatted message might repeatedly cause an MDB to fail and therefore cannot be delivered. Such a message is called a poison message. You can configure IBM MQ so that IBM MQ classes for JMS automatically transfers a poison message to another queue for further investigation or discards the message.

For details on how to handle poison messages, see “Handling poison messages in IBM MQ classes for JMS” on page 225.

Related concepts
Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client
Federal Information Processing Standards (FIPS) for AIX, Linux, and Windows

Related tasks

Configuring JMS resources in WebSphere Application Server

Examples of how to define the sharedSubscription property

You can define the sharedSubscription property of an activation specification within a WebSphere Liberty `server.xml` file. Alternatively, you can define the property within a message driven bean (MDB) using annotations.

Example: defining within a Liberty `server.xml` file

Within a WebSphere Liberty `server.xml` file, you define an activation specification as shown in the following example. This example creates a durable shared subscription to a queue manager on localhost/ port 1490.

```
<jmsActivationSpec id="SubApp/SubscribingEJB/SubscribingMDB" authDataRef="JMSConnectionAlias">
<properties.wmqJms hostName="localhost" port="1490" maxPoolDepth="5"
subscriptionName="MySubName"
subscriptionDurability="DURABLE" sharedSubscription="true"/>
</jmsActivationSpec>
```

Example: defining within an MDB

You can also define the sharedSubscription property within the MDB using annotations as shown in the following example:

```
@ActioncationConfigProperty(propertyName ="sharedSubscription",
propertyValue = "true")
```

The following example shows a piece of MDB code that uses the annotations method:

JMS 3.0

```
/**
 * Message-Driven Bean example using Annotations for configuration
 */
@MessageDriven(
  activationConfig = {
     @ActivationConfigProperty(
        propertyName = "destinationType", propertyValue = "jakarta.jms.Topic"),
     @ActivationConfigProperty(
        propertyName = "sharedSubscription", propertyValue = "TRUE"),
     @ActivationConfigProperty(
        propertyName = "destination", propertyValue = "JNDI_TOPIC_NAME")
   },
   mappedName = "Stock/IBM")
public class SubscribingMDB implements MessageListener {

   // Default constructor.
   public SubscribingMDB() {
   }

   // @see MessageListener#onMessage(Message)
   public void onMessage(Message message) {
      // implement business logic here
   }

}
```

JMS 2.0

```
/**
 * Message-Driven Bean example using Annotations for configuration
 */
@MessageDriven(
  activationConfig = {
     @ActivationConfigProperty(
        propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
     @ActivationConfigProperty(
        propertyName = "sharedSubscription", propertyValue = "TRUE"),
```

```
      @ActivationConfigProperty(
         propertyName = "destination", propertyValue = "JNDI_TOPIC_NAME")
   },
   mappedName = "Stock/IBM")
public class SubscribingMDB implements MessageListener {

   // Default constructor.
   public SubscribingMDB() {
   }

   // @see MessageListener#onMessage(Message)
   public void onMessage(Message message) {
      // implement business logic here
   }

}
```

Related concepts

Subscribers and subscriptions

Subscription durability

“Cloned and shared subscriptions” on page 315

There are two methods for giving multiple consumers access to the same subscription. These two methods are by using cloned subscriptions, or by using shared subscriptions.

Configuring the resource adapter for outbound communication

To configure outbound communication, define the properties of a ConnectionFactory object and an administered destination object.

Example of using outbound communication

When using outbound communication, an application running in an application server starts a connection to a queue manager, and then sends messages to its queues and receives messages from its queues in a synchronous manner. For example, the following servlet method, doGet(), uses outbound communication:

JMS 3.0

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {

...

// Look up ConnectionFactory and Queue objects from the JNDI namespace

     InitialContext ic = new InitialContext();
     ConnectionFactory cf = (jakarta.jms.ConnectionFactory) ic.lookup("myCF");
     Queue q = (jakarta.jms.Queue) ic.lookup("myQueue");

// Create and start a connection

     Connection c = cf.createConnection();
     c.start();

// Create a session and message producer

     Session s = c.createSession(false, Session.AUTO_ACKNOWLEDGE);
     MessageProducer pr = s.createProducer(q);

// Create and send a message

     Message m = s.createTextMessage("Hello, World!");
     pr.send(m);

// Create a message consumer and receive the message just sent

     MessageConsumer co = s.createConsumer(q);
     Message mr = co.receive(5000);

// Close the connection

      c.close();
}
```

JMS 2.0

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {

...

// Look up ConnectionFactory and Queue objects from the JNDI namespace

     InitialContext ic = new InitialContext();
     ConnectionFactory cf = (javax.jms.ConnectionFactory) ic.lookup("myCF");
     Queue q = (javax.jms.Queue) ic.lookup("myQueue");

// Create and start a connection

     Connection c = cf.createConnection();
     c.start();

// Create a session and message producer

     Session s = c.createSession(false, Session.AUTO_ACKNOWLEDGE);
     MessageProducer pr = s.createProducer(q);

// Create and send a message

     Message m = s.createTextMessage("Hello, World!");
     pr.send(m);

// Create a message consumer and receive the message just sent

     MessageConsumer co = s.createConsumer(q);
     Message mr = co.receive(5000);

// Close the connection

      c.close();
}
```

When the servlet receives an HTTP GET request, it retrieves a ConnectionFactory object and a Queue object from the JNDI namespace, and uses the objects to send a message to an IBM MQ queue. The servlet then receives the message that it has sent.

Resources needed for outbound communication

To configure outbound communication, define Java EE Connector Architecture (JCA) resources in the following categories:

- The properties of a ConnectionFactory object, which the application server uses to create a JMS ConnectionFactory object.
- The properties of an administered destination object, which the application server uses to create a JMS Queue object or JMS Topic object.

The way you define these properties depends on the administration interfaces provided by your application server. ConnectionFactory, Queue, and Topic objects created by the application server are bound into a JNDI namespace from where they can be retrieved by an application.

Typically, you define one ConnectionFactory object for each queue manager that applications might need to connect to. You define one Queue object for each queue that applications might need to access in the point-to-point domain. And you define one Topic object for each topic that applications might want to publish or subscribe to. A ConnectionFactory object can be domain independent. Alternatively, it can be domain-specific, a QueueConnectionFactory object for the point-to-point domain or a TopicConnectionFactory object for the publish/subscribe domain.

Tip: With JMS 2.0, a connection factory can be used to create both connections and contexts. As a result, it is possible to have a connection pool associated with a connection factory that contains a mixture of both connections and contexts. It is recommended that a connection factory is only used for creating connections or creating contexts. This ensures that the connection pool for that connection factory only contains objects of a single type, which makes the pool more efficient.

Properties of a ConnectionFactory object

Table 66 on page 455 lists the properties of a ConnectionFactory object. The application server uses these properties to create a JMS ConnectionFactory object.

Table 66. Properties of a ConnectionFactory object

Name of property	Type	Valid values (default value in bold)	Description
applicationName	String	• The invoking class name, if it is available, adjusted to be no longer than 28 characters. If it is not available, the string `WebSphere MQ Client for Java` is used.	The name by which an application is registered with the queue manager. This application name is shown by the **DISPLAY CONN MQSC/PCF** command (where the field is called **APPLTAG**) or in the IBM MQ Explorer **Application Connections** display (where the field is called **App name**).
brokerCCSubQueue	String	• **SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives nondurable subscription messages.
brokerControlQueue	String	• **SYSTEM.BROKER.CONTROL.QUEUE** • A queue name	The name of the broker control queue.
brokerPubQueue	String	• **SYSTEM.BROKER.DEFAULT.STREAM** • A queue name	The name of the queue where published messages are sent (the stream queue).
brokerQueueManager	String	• **"" (empty string)** • A queue manager name	The name of the queue manager on which the broker is running.
brokerSubQueue	String	• **SYSTEM.JMS.ND.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a nondurable message consumer receives messages. See BROKERSUBQ property for more information.
brokerVersion	String	• **unspecified** - After the broker has been migrated from V6 to V7, set this property so that RFH2 headers are no longer used. After migration this property is no longer relevant. • **V1** - To use an IBM MQ Publish/Subscribe broker. This value is the default value if TRANSPORT is set to BIND or CLIENT. • **V2** - To use a broker of IBM Integration Bus in native mode. This value is the default value if TRANSPORT is set to DIRECT or DIRECTHTTP.	The version of the broker being used.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
ccdtURL	String	• **null** • A uniform resource locator (URL)	A URL that identifies the name and location of the file containing the client channel definition table (CCDT) and specifies how the file can be accessed.
CCSID	String	• **819** • A coded character set identifier supported by the Java virtual machine (JVM)	The coded character set identifier for a connection.
channel	String	• **SYSTEM.DEF.SVRCONN** • The name of an MQI channel	The name of the MQI channel to use.
cleanupInterval	int	• **3 600 000** • A positive integer	The interval, in milliseconds, between background runs of the publish/subscribe cleanup utility.
cleanupLevel	String	• **SAFE** • NONE • STRONG • FORCE • NONDUR	The cleanup level for a broker-based subscription store.
clientID	String	• **null** • A client identifier	The client identifier for a connection.
cloneSupport	String	• **DISABLED** - Only one instance of a durable topic subscriber can run at a time. • ENABLED - Two or more instances of the same durable topic subscriber can run simultaneously, but each instance must run in a separate Java virtual machine (JVM).	Whether two or more instances of the same durable topic subscriber can run simultaneously.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
connectionNameList	String	• **localhost(1414)** • A string composed of items separated by commas where each item takes the format: `HOSTNAME(PORT)` where *HOSTNAME* is either a DNS name or an IP address.	A list of TCP/IP connection names used for outbound communications. **connectionNameList** supersedes the **hostname** and **port** properties. This property is used to reconnect to multi-instance queue managers. **connectionNameList** is similar in form to **localAddress**, but must not be confused with it. **localAddress** specifies the characteristics of the local communications, whereas **connectionNameList** specifies how to reach a remote queue manager.
failIfQuiesce	Boolean	• **true** • false	Whether calls to certain methods fail if the queue manager is in a quiescing state.
headerCompression	String	• **NONE** • SYSTEM - RLE message header compression is performed.	A list of the techniques that can be used for compressing header data on a connection.
hostName	String	• **localhost** • A host name • An IP address	The host name or IP address of the system on which the queue manager resides. The **hostname** and **port** properties are superseded by the **connectionNameList** property when it is specified.
localAddress	String	• **null** • A string in the format: `[ host_name ][(low_port [, high_port ])]` where *host_name* is a host name or IP address, *low_port* and *high_port* are TCP port numbers, and brackets denote an optional component	For a connection to a queue manager, this property specifies either or both of the following: • The local network interface to be used • The local port, or range of local ports, to be used **localAddress** is similar in form to **connectionNameList**, but must not be confused with it. **localAddress** specifies the characteristics of the local communications, whereas **connectionNameList** specifies how to reach a remote queue manager.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
messageCompression	String	• **NONE** • A list of one or more of the following values separated by blank characters: RLE ZLIBFAST ZLIBHIGH V 9.4.0 LZ4FAST V 9.4.0 LZ4HIGH	A list of the techniques that can be used for compressing message data on a connection.
messageSelection	String	• **CLIENT** • BROKER	Determines whether message selection is done by IBM MQ classes for JMS or by the broker. Message selection by the broker is not supported when **brokerVersion** has the value 1.
password	String	• **null** • A password	The default password to use when creating a connection to the queue manager.
pollingInterval	int	• **5000** • Any positive integer	If each message listener within a session has no suitable message on its queue, this value is the maximum interval, in milliseconds, that elapses before each message listener tries again to get a message from its queue. If it frequently happens that no suitable message is available for any of the message listeners in a session, consider increasing the value of this property. This property is relevant only if **TRANSPORT** has the value `BIND` or `CLIENT`.
port	int	• **1414** • A TCP port number	The port on which the queue manager listens. The **hostname** and **port** properties are superseded by the **connectionNameList** property when it is specified.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
providerVersion	string	• **unspecified** • A string in one of the following formats – V.R.M.F – V.R.M – V.R – V where V, R, M, and F are integer values greater than or equal to zero.	The version, release, modification level and fix pack of the queue manager to which the application intends to connect.
pubAckInterval	int	• **25** • A positive integer	The number of messages published by a publisher before IBM MQ classes for JMS requests an acknowledgment from the broker.
queueManager	String	• **"" (empty string)** • A queue manager name	The name of the queue manager to connect to.
receiveExit [3]	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the IBM MQ classes for Java interface, MQReceiveExit	Identifies a channel receive exit program, or a sequence of receive exit programs to be run in succession.
receiveExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel receive exit programs when they are called.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
rescanInterval	int	• **5000** • Any positive integer	When a message consumer in the point-to-point domain uses a message selector to select which messages it wants to receive, IBM MQ classes for JMS searches the IBM MQ queue for suitable messages in the sequence determined by the **MsgDeliverySequence** attribute of the queue. When IBM MQ classes for JMS finds a suitable message and delivers it to the consumer, IBM MQ classes for JMS resumes the search for the next suitable message from its current position in the queue. IBM MQ classes for JMS continues to search the queue in this way until it reaches the end of the queue, or until the interval of time in milliseconds, as determined by the value of this property, has expired. In each case, IBM MQ classes for JMS returns to the beginning of the queue to continue its search, and a new time interval commences.
securityExit [3]	String	• **null** • The fully qualified name of a class that implements the IBM MQ classes for Java interface, MQSecurityExit	Identifies a channel security exit program.
securityExitInit	String	• **null** • A string of user data	The user data that is passed to a channel security exit program when it is called.
sendCheckCount	int	• **0** • Any positive integer	The number of send calls to allow between checking for asynchronous put errors, within a single non-transacted JMS session.
sendExit [3]	String	• **null** • A string comprising one or more items separated by commas, where each item is the fully qualified name of a class that implements the IBM MQ classes for Java interface, MQSendExit	Identifies a channel send exit program, or a sequence of send exit programs to be run in succession.
sendExitInit	String	• **null** • A string comprising one or more items of user data separated by commas	The user data that is passed to channel send exit programs when they are called.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
shareConvAllowed	Boolean	• NO - A client connection cannot share its socket. • **YES** - A client connection can share its socket.	Whether a client connection can share its socket with other top-level JMS connections from the same process to the same queue manager, if the channel definitions match.
sparseSubscriptions	Boolean	• **false** - Subscriptions receive frequent matching messages. • true - Subscriptions receive infrequent matching messages. This value requires that the subscription queue can be opened for browse.	Controls the message retrieval policy of a TopicSubscriber object.
sslCertStores	String	• **null** • A string of one or more LDAP URLs separated by blanks. Each LDAP URL has the format: `ldap://host_name [: port ]` where *host_name* is a host name or IP address, *port* is a TCP port number, and brackets denote an optional component.	The Lightweight Directory Access Protocol (LDAP) servers that hold certificate revocation lists (CRLs) for use on a TLS connection.
sslCipherSuite	String	• **null** • The name of a CipherSuite	The CipherSuite to use for a TLS connection.
sslFipsRequired [2]	Boolean	• **false** • true	Whether a TLS connection must use a CipherSuite that is supported by the IBM Java JSSE FIPS provider (IBMJSSEFIPS).
sslPeerName	String	• **null** • A template for distinguished names	For a TLS connection, a template that is used to check the distinguished name in the digital certificate provided by the queue manager.
sslResetCount	int	• **0** • An integer in the range 0 - 999 999 999	The total number bytes sent and received by a TLS connection before the secret keys used by TLS are renegotiated.
sslSocketFactory	String	A string representing the fully qualified class name of a class providing an implementation of the javax.net.ssl.SSLSocketFactory interface, optionally including an argument to be passed to the constructor method, enclosed in parentheses.	Any connections established in the scope of the administered destination object use sockets obtained from this implementation of the SSLSocketFactory interface.

Table 66. Properties of a ConnectionFactory object (continued)			
Name of property	**Type**	**Valid values (default value in bold)**	**Description**
statusRefreshInterval	int	• **60000** • Any positive integer	The interval, in milliseconds, between refreshes of the long running transaction that detects when a subscriber loses its connection to the queue manager. This property is relevant only if **SUBSTORE** has the value QUEUE.
subscriptionStore	String	• **BROKER** • MIGRATE • QUEUE	Determines where IBM MQ classes for JMS stores persistent data about active subscriptions.
targetClientMatching	Boolean	• **true** • false	Whether a reply message, sent to the queue identified by the JMSReplyTo header field of an incoming message, has an MQRFH2 header only if the incoming message has an MQRFH2 header. You can also configure this property for an activation specification. For more information, see "Configuring the targetClientMatching property for an activation specification" on page 471.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
temporaryModel	String	• **SYSTEM.DEFAULT.MODEL.QUEUE** • SYSTEM.JMS.TEMPQ.MODEL • Any string	The name of the model queue from which JMS temporary queues are created. Use SYSTEM.DEFAULT.MODEL.QUEUE if both of the following statements are true: • Your application uses a temporary queue that will accept non-persistent messages. • Only one application will create a temporary queue on the queue manager that the ConnectionFactory points to at a time. Note that SYSTEM.DEFAULT.MODEL.QUEUE can only be opened by one application at a time. Use SYSTEM.JMS.TEMPQ.MODEL in the following situations: • When your application uses a temporary queue that will accept persistent messages. • If multiple applications can connect to the queue manager that the ConnectionFactory points to and those applications need to create temporary queues at the same time. Define a new model queue with the **DEFPSIST** attribute set to YES, and the **DEFSOPT** attribute set to SHARED in the following situation: • When your application uses a temporary queue that will accept non-persistent messages, and multiple applications will connect to the queue manager that the ConnectionFactory points to, and those applications need to create temporary queues at the same time. When the new model queue is created, set the **temporaryModel** property to the name of the new model queue.

Table 66. Properties of a ConnectionFactory object (continued)

Name of property	Type	Valid values (default value in bold)	Description
tempQPrefix	String	• **"" (empty string)** • A prefix that can be used to form the name of an IBM MQ dynamic queue. The rules for forming the prefix are the same as the rules for forming the contents of the **DynamicQName** field in an IBM MQ object descriptor, structure MQOD, but the last non-blank character must be an asterisk (*). If the value of the property is the empty string, IBM MQ classes for JMS uses the value AMQ.* when creating a dynamic queue.	The prefix that is used to form the name of an IBM MQ dynamic queue.
tempTopicPrefix	String	Any non-null string consisting only of valid characters for an IBM MQ topic string	When creating temporary topics, JMS generates a topic string of the form "TEMP/*TEMPTOPICPREFIX*/*unique_id* ", or if this property is left with the default value, just "TEMP/*unique_id*". Specifying a non-empty **TEMPTOPICPREFIX** allows specific model queues to be defined for creating the managed queues for subscribers to temporary topics created under this connection.

Table 66. Properties of a ConnectionFactory object (continued)			
Name of property	**Type**	**Valid values (default value in bold)**	**Description**
transportType	String	• **CLIENT** • BINDINGS • BINDINGS_THEN_CLIENT	Whether a connection to a queue manager uses client mode or bindings mode. If the value BINDINGS_THEN_CLIENT is specified, the resource adapter first tries to make a connection in bindings mode. If this connection attempt fails, the resource adapter then tries to make a client mode connection. z/OS If an activation specification that is running on a WebSphere Application Server for z/OS system has been configured to use the BINDINGS_THEN_CLIENT transport mode and a previously established connection is broken, then any reconnection attempts by the activation specification first attempt to use the BINDINGS transport mode. If the BINDINGS transport mode connection attempt is unsuccessful, the activation specification subsequently attempts a CLIENT transport mode connection.
username	String	• **null** • A user name	The default user name to use when creating a connection to a queue manager.
wildcardFormat	int	• CHAR- Recognizes character wildcards only, as used in broker version 1 • TOPIC - Recognizes topic level wildcards only, as used in broker version 2	Which version of wildcard syntax is to be used.

Notes:

1. For important information about using the sslFipsRequired property, see “Limitations of the IBM MQ resource adapter” on page 424.
2. For information on how to configure the resource adapter so that it can locate an exit, see “Configuring IBM MQ classes for JMS to use channel exits” on page 271.

The following example shows a typical set of properties of a ConnectionFactory object:

```
channel:         SYSTEM.DEF.SVRCONN
hostName:        192.168.0.42
port:            1414
queueManager:    ExampleQM
transportType:   CLIENT
```

Properties of an administered destination object

The application server uses the properties of an administered destination object to create a JMS Queue object or JMS Topic object.

Table 67 on page 466 lists the properties that are common to a Queue object and a Topic object.

Table 67. Properties that are common to a Queue object and a Topic object

Name of property	Type	Valid values (default value in bold)	Description
CCSID	String	• **1208** • A coded character set identifier supported by the Java virtual machine (JVM)	The coded character set identifier for the destination.
encoding	String	• **NATIVE** • A string of three characters: – The first character specifies the representation of binary integers: - *N* denotes normal encoding. - *R* denotes reverse encoding. – The second character specifies the representation of packed decimal integers: - *N* denotes normal encoding. - *R* denotes reverse encoding. – The third character specifies the representation of floating point numbers: - *N* denotes standard IEEE encoding. - *R* denotes reverse IEEE encoding. - *3* denotes zSeries encoding. NATIVE is equivalent to the string NNN.	The representation of binary integers, packed decimal integers, and floating point numbers for the destination.
expiry	String	• **APP** - The expiry time of a message is determined by the message producer. • UNLIM - A message never expires. • 0 - A message never expires. • A positive integer representing the expiry time of a message in milliseconds.	The expiry time of a message sent to the destination.
failIfQuiesce	String	• **true** • false	Whether an attempt to access the destination fails if the queue manager is in a quiescing state.

Table 67. Properties that are common to a Queue object and a Topic object (continued)

Name of property	Type	Valid values (default value in bold)	Description
messageBodyStyle	String	• **UNSPECIFIED** • JMS • MQ	You can set the **messageBodyStyle** property on JMS queues and topics: UNSPECIFIED(default) • When sending, IBM MQ classes for JMS generate and include an MQRFH2 header, depending on the value of WMQ_TARGET_CLIENT. • When receiving, IBM MQ classes for JMS set the JMS message properties according to values in the MQRFH2, if present. MQRFH2 is not presented as part of the JMS message body. JMS • When sending, IBM MQ classes for JMS automatically generates an MQRFH2 header and includes the header in the IBM MQ message. • When receiving, IBM MQ classes for JMS set the JMS message properties according to values in the MQRFH2, if present. MQRFH2 is not presented as part of the JMS message body. MQ • When sending, IBM MQ classes for JMS do not generate an MQRFH2. • When receiving, IBM MQ classes for JMS present the MQRFH2 as part of the JMS message body.

Table 67. Properties that are common to a Queue object and a Topic object (continued)

Name of property	Type	Valid values (default value in bold)	Description
persistence	String	• **APP** - The persistence of a message is determined by the message producer. • QDEF - The persistence of a message is determined by the **DefPersistence** attribute of the IBM MQ queue. • PERS - A message is persistent. • NON - A message is nonpersistent. • HIGH - The persistence of a message is determined by the **NonPersistentMessageClass** attribute of the IBM MQ queue according to the explanation in "JMS persistent messages" on page 245.	The persistence of a message sent to the destination.
priority	String	• **APP** - The priority of a message is determined by the message producer. • QDEF - The priority of a message is determined by the **DefPriority** attribute of the IBM MQ queue. • An integer in the range 0, lowest priority, to 9, highest priority.	The priority of a message sent to the destination.
putAsyncAllowed	String	• QUEUE - Determine whether asynchronous puts are allowed by referring to the queue definition. • TOPIC - Determine whether asynchronous puts are allowed by referring to the topic definition. • DESTINATION - Determine whether asynchronous puts are allowed by referring to the queue or topic definition. • DISABLED - Asynchronous puts are not allowed. • ENABLED - Asynchronous puts are allowed.	Whether message producers are allowed to use asynchronous puts to send messages to this destination.
readAheadAllowed	int	• **DESTINATION** - Determine whether read ahead is allowed by referring to the queue or topic definition. • DISABLED - Read ahead is not allowed. • ENABLED - Read ahead is allowed. • QUEUE - Determine whether read ahead is allowed by referring to the queue definition. • TOPIC - Determine whether read ahead is allowed by referring to the topic definition.	Whether message consumers and queue browsers are allowed to use read ahead to get nonpersistent messages from the destination into an internal buffer before receiving them.

Table 67. Properties that are common to a Queue object and a Topic object (continued)			
Name of property	**Type**	**Valid values (default value in bold)**	**Description**
receiveCCSID	int	• **0** - Use JVM `Charset.defaultCharset` • 1208 - UTF-8 • A supported coded character set identifier	Destination property that sets the target CCSID for queue manager message conversion. The value is ignored unless **`receiveConversion`** is set to QMGR.
receiveConversion	String	• **CLIENT_MSG** • QMGR	Destination property that determines whether data conversion is going to be performed by the queue manager.
targetClient	String	• **JMS** - The target of a message is a JMS application. • MQ - The target of a message is a non-JMS IBM MQ application.	Whether the target of a message sent to the destination is a JMS application. A message with a target that is a JMS application contains an MQRFH2 header.

Table 68 on page 469 lists the properties that are specific to a Queue object.

Table 68. Properties that are specific to a Queue object			
Name of property	**Type**	**Valid values (default value in bold)**	**Description**
baseQueueManagerName	String	• **"" (empty string)** • A queue manager name	The name of the queue manager that owns the underlying IBM MQ queue.
baseQueueName	String	• **"" (empty string)** • A queue name	The name of the underlying IBM MQ queue.

Table 69 on page 469 lists the properties that are specific to a Topic object.

Table 69. Properties that are specific to a Topic object			
Name of property	**Type**	**Valid values (default value in bold)**	**Description**
baseTopicName	String	• **"" (empty string)** • A topic name	The name of the underlying topic.
brokerCCDurSubQueue >	String	• **SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a connection consumer receives durable subscription messages.

Table 69. Properties that are specific to a Topic object (continued)

Name of property	Type	Valid values (default value in bold)	Description
brokerDurSubQueue	String	• **SYSTEM.JMS.D.SUBSCRIBER.QUEUE** • A queue name	The name of the queue from which a durable topic subscriber receives messages. See the **BROKEDURRSUBQ** property in the IBM MQ Explorer documentation for more information.
brokerPubQueue	String	• **Not set** • A queue name	The name of the queue where published messages are sent (the stream queue). The value of this property overrides the value of the **brokerPubQueue** property of the ConnectionFactory object. However, if you do not set the value of this property, the value of the **brokerPubQueue** property of the ConnectionFactory object is used instead.
brokerPubQueueManager	String	• **"" (empty string)** • A queue manager name	The name of the queue manager that owns the queue where messages published on the topic are sent.
brokerVersion	String	• **Not set** • 1 • 2	The version of the broker being used. The value of this property overrides the value of the **brokerVersion** property of the ConnectionFactory object. However, if you do not set the value of this property, the value of the **brokerVersion** property of the ConnectionFactory object is used instead.

The following example shows a set of properties of a Queue object:

```
expiry:               UNLIM
persistence:          QDEF
baseQueueManagerName: ExampleQM
baseQueueName:        SYSTEM.JMS.TEMPQ.MODEL
```

The following example shows a set of properties of a Topic object:

```
expiry:               UNLIM
persistence:          NON
baseTopicName:        myTestTopic
```

Related tasks

Specifying that only FIPS-certified CipherSpecs are used at run time on the MQI client

Configuring JMS resources in WebSphere Application Server

Related reference

Federal Information Processing Standards (FIPS) for AIX, Linux, and Windows

Configuring the targetClientMatching property for an activation specification

You can configure the **targetClientMatching** property for an activation specification so that an MQRFH2 header is included on reply messages when request messages do not contain an MQRFH2 header. This means that any message properties that an application defines on a reply message are included when the message is sent.

About this task

If a message-driven bean (MDB) application consumes messages that do not contain an MQRFH2 header, through an IBM MQ JCA resource adapter activation specification, and subsequently sends reply messages to the JMS Destination created from the JMSReplyTo field of the request message, the reply messages must include an MQRFH2 header, even if the request messages do not, otherwise any message properties that the application has defined on a reply message are lost.

The **targetClientMatching** property defines whether a reply message, sent to the queue identified by the JMSReplyTo header field of an incoming message, has an MQRFH2 header only if the incoming message has an MQRFH2 header. You can configure this property for an activation specification, in both WebSphere Application Server traditional and WebSphere Liberty.

If you set the value of the **targetClientMatching** property to `false`, an MQRFH2 header can be included in a reply message sent to a JMS Destination created from the JMSReplyTo header of an incoming request message that does not contain an MQRFH2. This is because the **targetClient** property on the JMS Destination is set to the value `0`, which means that messages contain an MQRFH2 header. The presence of the MQRFH2 header in the outbound message permits the storage of user defined message properties on the message when sent to the IBM MQ queue.

If the **targetClientMatching** property is set to `true` and a request message does not include an MQRFH2 header, an MQRFH2 header is not included in the reply message.

Procedure

- In WebSphere Application Server traditional, use the administration console to define the **targetClientMatching** property as a custom property on the IBM MQ activation specification:
  a) In the navigation pane, click **Resources -> JMS ->Activation specifications**.
  b) Select the name of the activation specification that you want to view or change.
  c) Click **Custom properties -> New** and then enter the details of the new custom property.
  Set the name of the property to `targetClientMatching`, the type to `java.lang.Boolean` and the value to `false`.
- In WebSphere Liberty, specify the **targetClientMatching** property on the definition of an activation specification within the `server.xml`.
  For example:

```
<jmsActivationSpec id="SimpleMDBApplication/SimpleEchoMDB/SimpleEchoMDB">
<properties.wmqJms destinationRef="MDBRequestQ"
queueManager="MY_QMGR" transportType="BINDINGS" targetClientMatching="false"/>
<authData password="********" user="tom"/>
</jmsActivationSpec>
```

Related concepts

"Creating destinations in a JMS application" on page 212

Instead of retrieving destinations as administered objects from a Java Naming and Directory Interface (JNDI) namespace, a JMS application can use a session to create destinations dynamically at run time. An application can use a uniform resource identifier (URI) to identify an IBM MQ queue or a topic and, optionally, to specify one or more properties of a Queue or Topic object.

“Configuring the resource adapter for outbound communication” on page 453
To configure outbound communication, define the properties of a ConnectionFactory object and an administered destination object.

IBM MQ message-driven bean pause in WebSphere Liberty

The **maxSequentialDeliveryFailures** property for an activation specification defines the maximum number of sequential message delivery failures to a message-driven bean (MDB) instance that the resource adapter tolerates before pausing the MDB.

Before you begin

You need to be aware of the set of events that might cause an MDB to pause in WebSphere Liberty. The resource adapter considers any one of the following as a message delivery failure:

- An unchecked exception being thrown from the **onMessage** method of the MDB.
- A `JMSException` occurring in the processing of the resource adapter, prior to delivering the message to the MDB.
- A `JMSException` occurring in the processing of the resource adapter, post delivering the message to the MDB.
- The XA transaction, or local transaction, used to consume the message being rolled-back.
- No thread being available in the application server to deliver the message to the MDB.

About this task

The default value of the **maxSequentialDeliveryFailures** property is *-1*, which means that the MDB is never paused. Any other negative value is treated the same as *-1*. A value of:

- *0* means that the MDB pauses on the first error
- *1* means that the MDB pauses on two sequential errors
- *2* means that the MDB pauses on three sequential errors, and so on

You can configure this property for an activation specification, only in WebSphere Liberty, and when the level of Liberty is 18.0.0.4, or higher.

Attention: If you set the attribute to a non-default value in any application server environment other than Liberty, the value will be ignored and a warning message written to the log.

In addition, it is possible to install the IBM MQ resource adapter into WebSphere Liberty as a generic resource adapter. Doing this disables all of the IBM MQ and WebSphere Application Server integration capabilities, and prevents the resource adapter from being able to detect that it is running in Liberty. Therefore, setting **maxSequentialDeliveryFailures** to be greater than or equal to *0* is not supported, and results in a warning message in the log.

Procedure

- In WebSphere Liberty, specify the **maxSequentialDeliveryFailures** property on the definition of an activation specification within the `server.xml`.

  For example:

```
<jmsActivationSpec>
     <properties.wmqJms destinationRef="jndi/MDBQ"
                        transportType="BINDINGS"
                        queueManager="MQ21"
                        maxSequentialDeliveryFailures="1"/>
</jmsActivationSpec>
```

Related concepts

“Configuring the resource adapter for outbound communication” on page 453

To configure outbound communication, define the properties of a ConnectionFactory object and an administered destination object.

Verifying the resource adapter installation

The installation verification test (IVT) program for the IBM MQ resource adapter is supplied as an EAR file. To use the program, you must deploy it and define some objects as JCA resources.

About this task

The installation verification test (IVT) program is supplied as an enterprise archive (EAR) file called `wmq.jakarta.jmsra.ivt.ear` (Jakarta Messaging 3.0) or `wmq.jmsra.ivt.ear` (JMS 2.0). This file is installed with IBM MQ classes for JMS in the same directory as the IBM MQ resource adapter RAR file, `wmq.jakarta.jmsra.rar` (Jakarta Messaging 3.0) or `wmq.jmsra.rar` (JMS 2.0). For information about where these files are installed, see "Installing the IBM MQ resource adapter" on page 428.

You must deploy the IVT program on your application server. The IVT program includes a servlet and an MDB that tests that a message can be sent to, and received from, an IBM MQ queue. You can use the IVT program to verify that the IBM MQ resource adapter has been correctly configured to support distributed transactions. If you are deploying the IBM MQ resource adapter in an non-IBM application server, IBM Service might ask you to demonstrate the IVT working to validate that your application server is correctly configured.

Before you can run the IVT program, you must define a ConnectionFactory object, a Queue object and possibly an Activation Specification object as JCA resources, and ensure that your application server creates JMS objects from these definitions and binds them into a JNDI namespace. You can choose the properties of the objects to match the host and port settings of your own QueueManager, but the following set of properties is a simple example:

```
ConnectionFactory object:
channel:              SYSTEM.DEF.SVRCONN
hostName:             localhost
port:                 1550
queueManager:         QM1
transportType:        CLIENT
Queue object:
baseQueueManagerName: QM1
baseQueueName:        TEST.QUEUE
```

The mechanism used to define the ConnectionFactory, Queue and Activation Specification objects varies depending on your application server. For example, to set these properties within WebSphere Liberty, add the following entries to the application server's `server.xml` file:

```
JMS 3.0<!-- IVT Connection factory -->
<jmsQueueConnectionFactory connectionManagerRef="ConMgrIVT" jndiName="IVTCF">
   <properties.wmqJms channel="SYSTEM.DEF.SVRCONN" hostname="localhost" port="1550"
transportType="CLIENT"/>
</jmsQueueConnectionFactory>
<connectionManager id="ConMgrIVT" maxPoolSize="10"/>

<!-- IVT Queues -->
<jmsQueue id="IVTQueue" jndiName="IVTQueue">
   <properties.wmqJms baseQueueName="TEST.QUEUE"/>
</jmsQueue>

<!-- IVT Activation Spec -->
<jmsActivationSpec id="wmq.jakarta.jmsra.ivt/WMQ_IVT_MDB/WMQ_IVT_MDB">
   <properties.wmqJms destinationRef="IVTQueue"
transportType="CLIENT"
queueManager="QM1"
hostName="localhost"
port="1550"
maxPoolDepth="1"/>
</jmsActivationSpec>
```

```
JMS 2.0<!-- IVT Connection factory -->
<jmsQueueConnectionFactory connectionManagerRef="ConMgrIVT" jndiName="IVTCF">
```

```
   <properties.wmqJms channel="SYSTEM.DEF.SVRCONN" hostname="localhost" port="1550"
transportType="CLIENT"/>
</jmsQueueConnectionFactory>
<connectionManager id="ConMgrIVT" maxPoolSize="10"/>

<!-- IVT Queues -->
<jmsQueue id="IVTQueue" jndiName="IVTQueue">
   <properties.wmqJms baseQueueName="TEST.QUEUE"/>
</jmsQueue>

<!-- IVT Activation Spec -->
<jmsActivationSpec id="wmq.jmsra.ivt/WMQ_IVT_MDB/WMQ_IVT_MDB">
   <properties.wmqJms destinationRef="IVTQueue"
transportType="CLIENT"
queueManager="QM1"
hostName="localhost"
port="1550"
maxPoolDepth="1"/>
</jmsActivationSpec>
```

By default, the IVT program expects a ConnectionFactory object to be bound in the JNDI namespace with the name jms/ivt/IVTCF and a Queue object to be bound with the name jms/ivt/IVTQueue. You can use different names, but if you do, you must enter the names of the objects on the initial page of the IVT program and modify the EAR file appropriately.

After you have deployed the IVT program, and the application server has created the JMS objects and bound them into the JNDI namespace, you can start the IVT program by completing the following steps.

Procedure

1. Start the IVT program by entering a URL in the following format into your web browser:

   ```
   http://app_server_host: port/WMQ_IVT/
   ```

   where *app_server_host* is the IP address or host name of the system on which your application server is running, and *port* is the number of the TCP port on which the application server is listening. Here is an example:

   ```
   http://localhost:9080/WMQ_IVT/
   ```

   Here is an example of the initial page displayed by the IVT program.

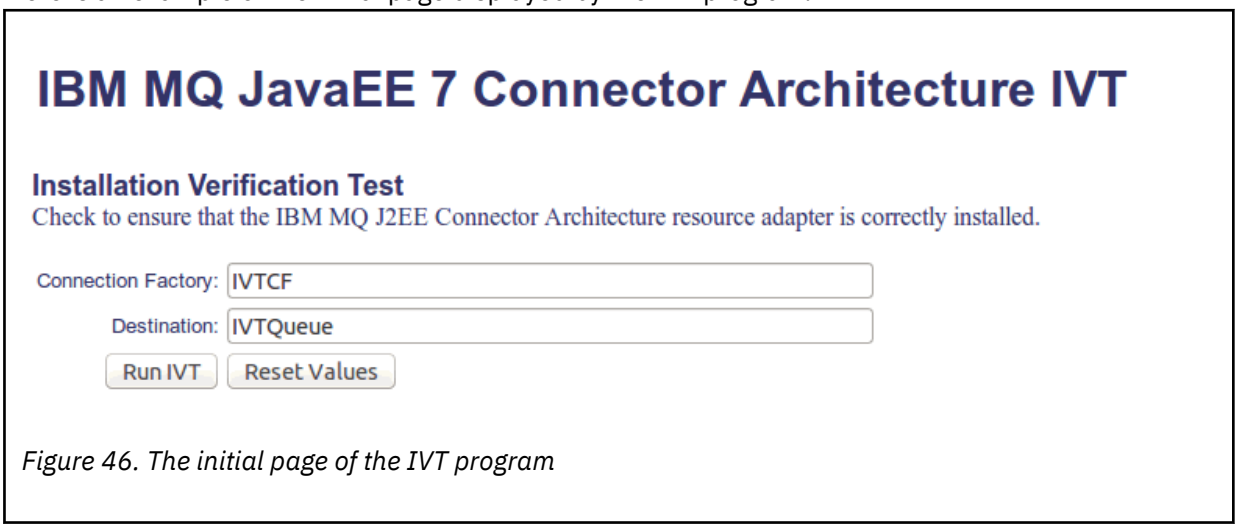
   

   Figure 46. The initial page of the IVT program

2. To run the test, click **Run IVT**.

   Here is an example of the page that is displayed if the IVT is successful.

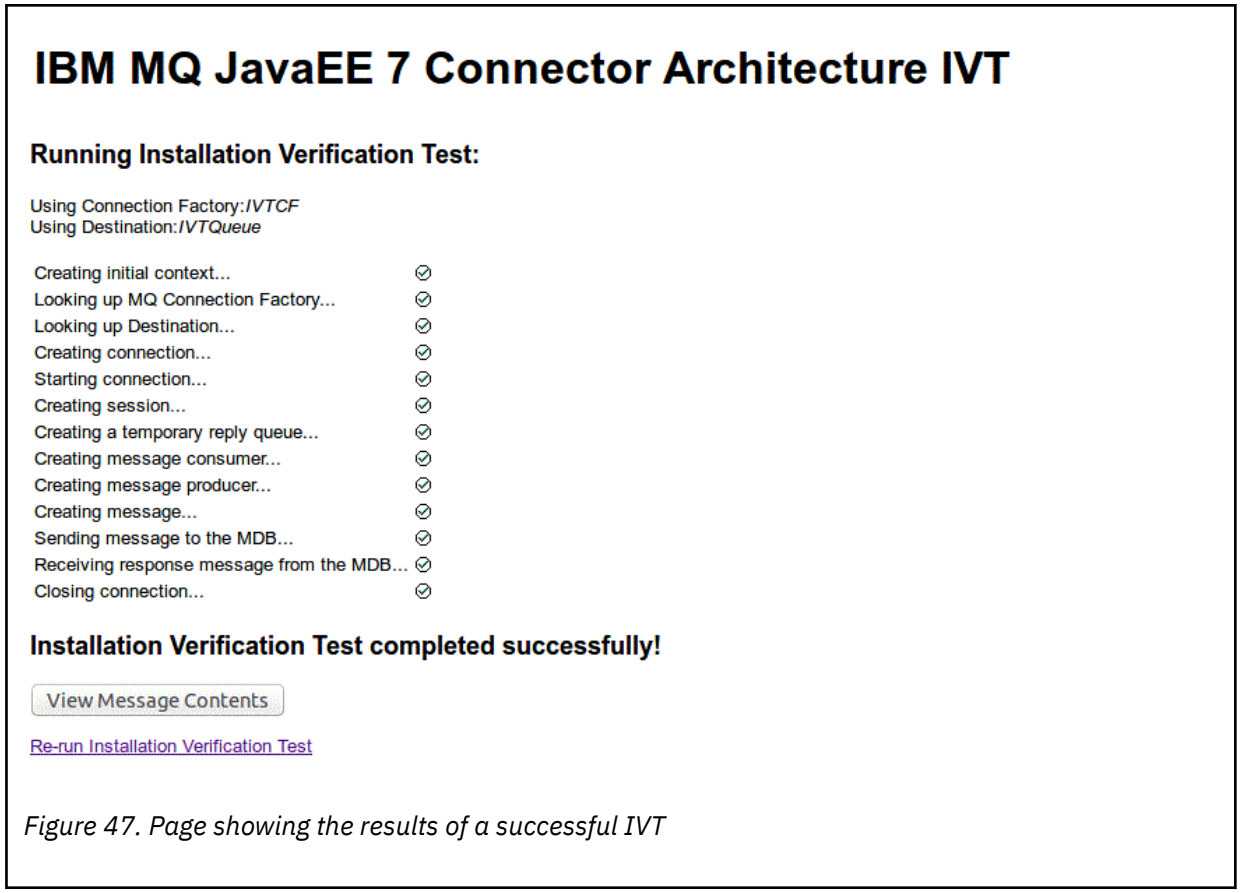


Figure 47. Page showing the results of a successful IVT

Here is an example of the page that is displayed if the IVT fails. To obtain further information about the cause of the failure, click **View Stack Trace**.

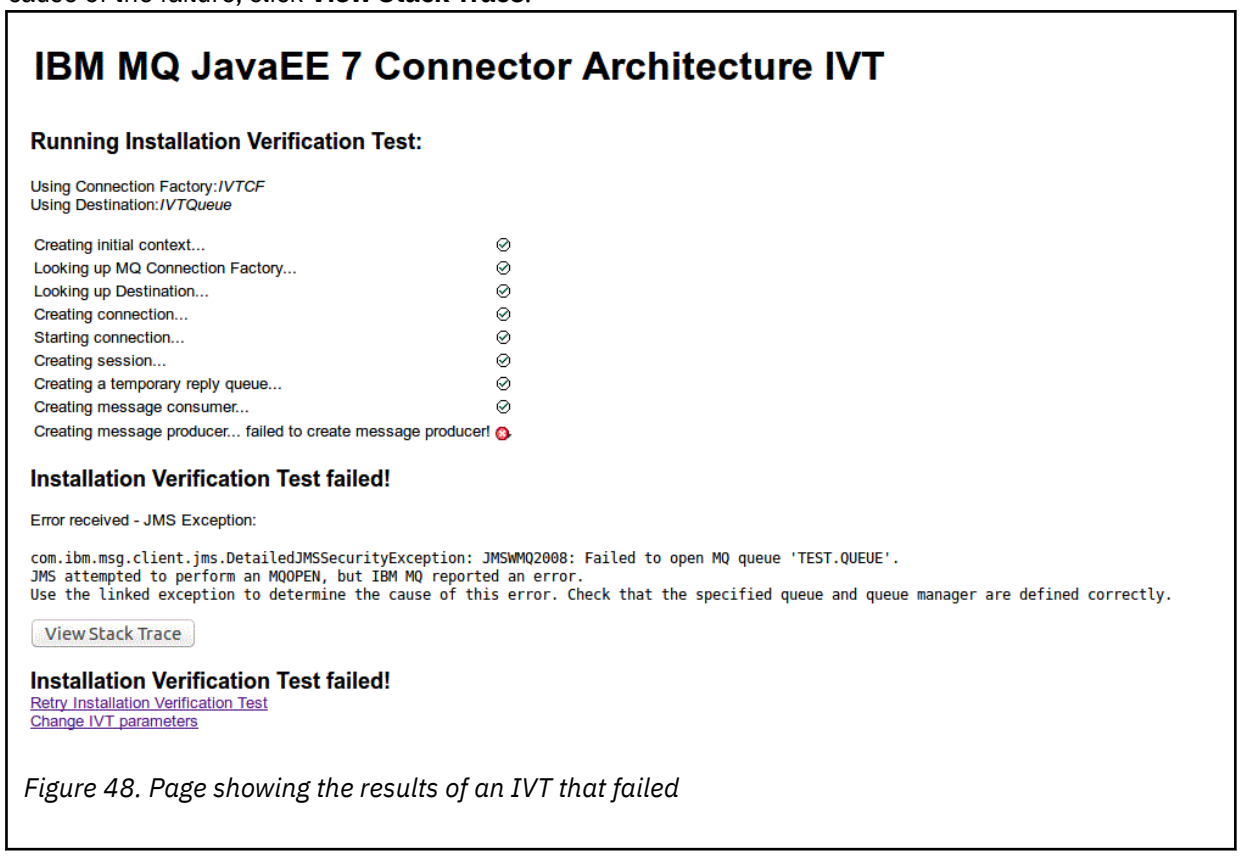


Figure 48. Page showing the results of an IVT that failed

Windows Installing and testing the resource adapter in GlassFish Server

To install the IBM MQ resource adapter in GlassFish Server on a Windows operating system, you must first create and start a domain. You can then deploy and configure the resource adapter, and deploy and run the installation verification test (IVT) application.

Before you begin

- These instructions are for GlassFish Server version 4.
- This version of GlassFish Server does not support Jakarta EE.

About this task

This task assumes that you have a running GlassFish Server application server, and that you are familiar with standard administration tasks for it. This task also assumes that you have an IBM MQ installation on your local system and that you are familiar with standard administration tasks.

Note: In order to complete the following task steps, you must have a functioning IBM MQ installation, with the following objects configured:

- A queue manager called QM, that is started on port 1414, that uses channel SYSTEM.DEF.SVRCONN, and that connects using Client transport.
- A queue called Q1.

Procedure

1. Start the GlassFish Server **asadmin** shell program.
   a) Open the Windows command line and navigate to the *GlassFish*/bin directory, where *GlassFish* is the directory where GlassFish Server version 4 is installed.
   b) Enter the command **asadmin** in the command line.
   The **asadmin** command opens a shell program in the command line that enables you to create a new domain.

   GlassFish Server version 4 is started on your system.
2. Create, and then start a domain.
   a) Use the **create-domain** command, specifying the port and domain name, to create a new domain. Enter the following command on the command line:

   ```
   create-domain --adminport port domain_name
   ```

   where *port* is the port number, and *domain_name* is the name you want the domain to use.

   Note: The **create-domain** command has many optional parameters associated with it. However, for this task you need the `--adminport` parameter only. For more information, see the product documentation for GlassFish Server version 4.

   If the port that you specified is in use, the following message appears:

   `Port for domain_name port is in use`

   If the domain name you specified is in use, you receive a message telling you that your specified name is already in use, as well as a list of all domain names that are currently unavailable.
   b) When prompted to input a user name and password, enter the credentials to be used to log on to the application server through a web browser.
   If the command completes successfully, a message summarizing the domain creation is displayed on the command line, including the message `Command create-domain executed successfully.`

   You have successfully created a domain.

c) Start your domain by entering the following command into the command line:

```
start-domain domain_name
```

where *domain_name* is the domain name you previously specified.

3. Use a web browser to access GlassFish application server.
   a) In the address bar of a web browser enter the following command:

```
localhost:port
```

where *port* is the port that you specified earlier when creating your domain.

The GlassFish Console is displayed.

   b) When the GlassFish Console has loaded, and you are prompted for a user name and password, enter the credentials that you specified in step 2b.
4. Upload the resource adapter to GlassFish Server 4.
   a) On the toolbar **Common Tasks** select the **Applications** menu item to display the **Applications** page.
   b) Click the **Deploy** button, to open the **Deploy Applications or Modules** page.
   c) Click the **Browse** button, then navigate to the location of the `wmq.jmsra.rar` file. Select the file then click **OK**.
5. Create a connection pool.
   a) On the toolbar, under **Resources**, select the **Connectors** menu item.
   b) Then select **Connector Connection Pools** menu item, to open the **Connector Connection Pools** page.
   c) Click **New** to open the **New Connector Connection Pool (Step 1 of 2)** page.
   d) On **New Connector Connection Pool (Step 1 of 2)** page, input the pool name as jms/ivt/IVTCF-Connection-Pool into the **Pool Name** field.
   e) In the **Resource Adapter** field select `wmq.jmsra`.
   f) In the **Connection Definition** field, enter `javax.jms.ConnectionFactory`.
   g) Select **Next**, then select **Finish**.
6. Create the connector resources.
   a) On the toolbar, under the **Connectors** menu, select the **Connector Resource** option, to open the **Connector Resources** page.
   b) Select **New**, to open the **New Connector Resource** page.
   c) In the **JNDI Name** field, enter `IVTCF`.
   d) In the **Pool Name** field, enter `jms/ivt/IVTCF-Connection-Pool`.
   e) Leave all other fields empty.
   f) For each of the following property/value pairs, click **Add Property**, and enter the property name and the value as shown in the following example:
      - name: host; value: localhost
      - name: port; value 1414
      - name: channel; value: SYSTEM.DEF.SVRCONN
      - name: queueManager; value: QM
      - name: transportType; value: CLIENT

      Note: Make sure that you use the correct values for your own configuration settings, which might be different from the ones shown in this example.

g) In the toolbar, under **Connectors**, select the **Admin Object Resources** menu item, to open the **Admin Object Resources** page.

h) In the **Admin Object Resources** page, click **New** to open the **New Admin Object Resource** page.

i) In the **JNDI Name** field, enter `IVTQueue`.

j) In the **Resource Adapter** field, enter `wmq.jmsra`.

k) In the **Resource Type** field, enter `javax.jms.Queue`.

l) Leave the **Class Name** field as it is.

m) For each of the following property/value pairs, click **Add Property**, and enter the property name and the value as shown in the following example:

- name: name; value: IVTQueue
- name: baseQueueManagerName; value QM
- name: baseQueueName; value: Q1

Note: Make sure that you use the correct values for your own configuration settings, which might be different from the ones shown in this example.

n) Click **OK**.

o) Select the **Enabled** checkbox, then click **Enable**.

7. Deploy the EAR file `wmq.jmsra.ivt.ear` into GlassFish Server.
   a) Click the **Applications** option in the toolbar to display the **Applications** page.
   b) Click **Deploy** to add the IVT application.
   c) In the **Location** field navigate to, and select, the `wmq.jmsra.ivt.ear`.
   d) In the **Virtual Servers** field, select **server**, and then click **OK**.
8. Launch the IVT program.
   a) Click the **Applications** option in the toolbar to display the **Applications** page.
   b) Click on `wmq.jmsra.ivt` in the Deployed Applications table.
   c) Click the **Launch** button, in the Modules and Components table.
   d) Select the http: link.
   e) Click **Run IVT**.

   You have launched the IVT program, and if you are successful, the following output is displayed:

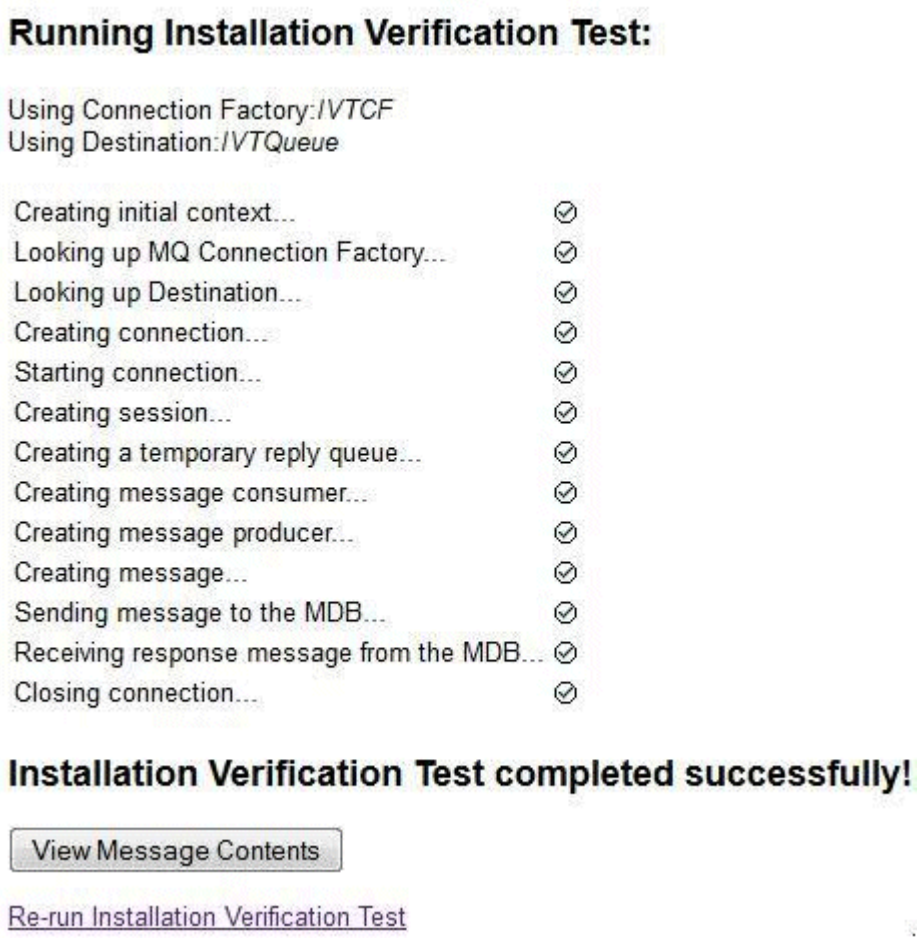


Figure 49. Successful IVT output

Installing and testing the resource adapter in WildFly

If you are installing the IBM MQ resource adapter in WildFly V10, you must first make some configuration file changes to add a subsystem definition for the IBM MQ resource adapter. You can then deploy the resource adapter and test it by installing and running the installation verification test (IVT) application.

Before you begin

- These instructions are for WildFly V10.
- This version of WildFly does not support Jakarta EE.

About this task

This task assumes that you have a running WildFly application server, and that you are familiar with standard administration tasks for it. This task also assumes that you have an IBM MQ installation and that you are familiar with standard administration tasks.

Procedure

1. Create an IBM MQ queue manager called ExampleQM, and set it up as described in "Configuring a queue manager to accept client connections on Multiplatforms" on page 1028.

   When setting up the queue manager, note the following points:

   - The listener must be started on port 1414.
   - The channel to be used is called SYSTEM.DEF.SVRCONN.

- The queue used by the IVT application is named TEST.QUEUE.

The model queue SYSTEM.DEFAULT.MODEL.QUEUE also needs to be granted DSP and PUT authority so this application can create a temporary reply queue.

2. Edit the configuration file *WildFly_Home*/standalone/configuration/standalone-full.xml and add the following subsystem:

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:4.0">
    <resource-adapters>
        <resource-adapter id="wmq.jmsra">
            <archive>
                wmq.jmsra.rar
            </archive>
            <transaction-support>NoTransaction</transaction-support>
            <connection-definitions>
                <connection-definition class-
name="com.ibm.mq.connector.outbound.ManagedConnectionFactoryImpl"
                                       jndi-name="java:jboss/jms/ivt/IVTCF" enabled="true"
use-java-context="true"
                                       pool-name="IVTCF">
                    <config-property name="channel">SYSTEM.DEF.SVRCONN
                    </config-property>
                    <config-property
                                      name="hostName">localhost
                    </config-property>
                    <config-property name="transportType">
                        CLIENT
                    </config-property>
                    <config-property name="queueManager">
                        ExampleQM
                    </config-property>
                    <config-property name="port">
                        1414
                    </config-property>
                </connection-definition>
                <connection-definition class-
name="com.ibm.mq.connector.outbound.ManagedConnectionFactoryImpl"
                                       jndi-name="java:jboss/jms/ivt/JMS2CF" enabled="true"
use-java-context="true"
                                       pool-name="JMS2CF">
                    <config-property name="channel">
                        SYSTEM.DEF.SVRCONN
                    </config-property>
                    <config-property name="hostName">
                        localhost
                    </config-property>
                    <config-property name="transportType">
                        CLIENT
                    </config-property>
                    <config-property name="queueManager">
                        ExampleQM
                    </config-property>
                    <config-property name="port">
                        1414
                    </config-property>
                </connection-definition>
            </connection-definitions>
        <admin-objects>
            <admin-object class-name="com.ibm.mq.connector.outbound.MQQueueProxy"
                          jndi-name="java:jboss/jms/ivt/IVTQueue" pool-name="IVTQueue">
                <config-property name="baseQueueName">
                    TEST.QUEUE
                </config-property>
                </admin-object>
            </admin-objects>
        </resource-adapter>
    </resource-adapters>
</subsystem>
```

3. Deploy the resource adapter to your server by copying the wmq.jmsra.rar file into the directory *WildFly_Home*/standalone/deployments.
4. Deploy the IVT application by coping the wmq.jmsra.ivt.ear file into the directory *WildFly_Home*/standalone/deployments.
5. Start the application server, by bringing up a command prompt, navigating to the directory *WildFly_Home*/bin and running the command:

```
standalone.bat -c standalone-full.xml
```

6. Run the IVT application.

   For more information, see “Verifying the resource adapter installation” on page 473. For WildFly, the default URL is `http://localhost:8080/WMQ_IVT/`.

Using IBM MQ and WebSphere Application Server together

Through the IBM MQ messaging provider in WebSphere Application Server, Java Message Service (JMS) messaging applications can use your IBM MQ system as an external provider of JMS messaging resources.

About this task

Applications that are written in Java and are running under WebSphere Application Server can use the Java Message Service (JMS) specification to perform messaging. Messaging in this environment can be provided by an IBM MQ queue manager.

A benefit of using an IBM MQ queue manager is that connecting JMS applications can participate fully in the functionality of an IBM MQ network, which allows the applications to exchange messages with queue managers that are running on a multitude of platforms.

Applications can use either the *client transport* or *bindings transport* for the queue connection factory object. For bindings transport, the queue manager must exist locally to the application that requires a connection.

By default, JMS messages that are held on IBM MQ queues use an MQRFH2 header to hold some of the JMS message header information. Many legacy IBM MQ applications cannot process messages with these headers, and require their own characteristic headers, for example the MQCIH for CICS Bridge, or MQWIH for IBM MQ Workflow applications. For more information about these special considerations, see Mapping JMS messages onto IBM MQ messages.

Related tasks

Configuring JMS resources in WebSphere Application Server

Configuring the application server to use the latest resource adapter maintenance level

Using WebSphere Application Server with IBM MQ

IBM MQ and IBM MQ for z/OS can be used with, or as an alternative to, the default messaging provider that is included with WebSphere Application Server.

The IBM MQ messaging provider is installed as part of WebSphere Application Server. This includes a version of the IBM MQ resource adapter, and the IBM MQ Extended Transactional Client functionality, which allows the queue manager to participate in XA transactions managed by the application server. Using the resource adapter, message-driven beans can be configured to use either activation specifications or listener ports.

For the application server to be supported, the IBM MQ resource adapter installation verification test program needs to be deployed into the application server and run successfully. After the IBM MQ resource adapter installation verification test program has been run successfully, the IBM MQ resource adapter can connect to any supported IBM MQ queue manager.

JMS connections from WebSphere Application Server to IBM MQ

Before considering the levels of IBM MQ that can be used with WebSphere Application Server, it is important to understand how Java Message Service (JMS) applications running inside the application server can connect to IBM MQ queue managers.

JMS applications that need to access the resources of an IBM MQ queue manager can do so by using one of the following transport types:

BINDINGS

This transport can be used when the application server and the queue manager are installed on the same machine and operating system image. When using BINDINGS mode, all of the communication between the two products is done using Inter-Process Communication (IPC).

The IBM MQ messaging provider does not include the native libraries required to connect to an IBM MQ queue manager in BINDINGS mode. In order to use a BINDINGS mode connection, IBM MQ must be installed onto the same machine as the application server, and the resource adapter's native library path must be configured to point to the IBM MQ directory where these libraries are located. For more information, see the WebSphere Application Server product documentation:

- For WebSphere Application Server traditional, see Configuring the IBM MQ messaging provider with native libraries.
- For WebSphere Liberty, see Deploying JMS applications to Liberty to use the IBM MQ messaging provider.

z/OS On z/OS, if you want to connect a WebSphere Application Server connection factory to an IBM MQ queue manger in bindings mode, you must specify the correct IBM MQ libraries in the WebSphere Application Server STEPLIB concatenation. For more information, see IBM MQ libraries and the WebSphere Application Server for z/OS STEPLIB in the WebSphere Application Server product documentation.

CLIENT

The client transport uses TCP/IP to communicate between WebSphere Application Server and IBM MQ. As well as being used when the application server and queue manager are located on different machines, CLIENT mode can also be used when the two products are installed on the same machine and operating system image.

JMS applications can also specify a transport type of BINDINGS_THEN_CLIENT. When this transport type is used, the application will initially attempt to connect to the queue manager using BINDINGS mode - if it is unable to do so, it will try the CLIENT transport.

How to find which version of the IBM MQ resource adapter is installed inside WebSphere Application Server

For information about which version of the IBM MQ resource adapter is installed inside WebSphere Application Server, see the technote Which version of WebSphere MQ Resource Adapter (RA) is shipped with WebSphere Application Server?.

You can use the following Jython and JACL commands to determine the level of the resource adapter that WebSphere Application Server is currently using:

Jython

```
wmqInfoMBeansUnsplit = AdminControl.queryNames("WebSphere:type=WMQInfo,*")
wmqInfoMBeansSplit = AdminUtilities.convertToList(wmqInfoMBeansUnsplit)
for wmqInfoMBean in wmqInfoMBeansSplit: print wmqInfoMBean; print
AdminControl.invoke(wmqInfoMBean, 'getInfo', '')
```

Note: You need to click **Return** twice after entering this command in order to run it.

JACL

```
set wmqInfoMBeans [$AdminControl queryNames WebSphere:type=WMQInfo,*]
foreach wmqInfoMBean $wmqInfoMBeans {
puts $wmqInfoMBean;
puts [$AdminControl invoke $wmqInfoMBean getInfo [] []]
}
```

Updating the resource adapter

Updates to the IBM MQ resource adapter that is installed with the application server are included in WebSphere Application Server Fix Packs. Updating the IBM MQ resource adapter using the **Update resource adapter...** facility in the WebSphere Application Server Administrative Console is not recommended, since doing so will mean that updates provided in WebSphere Application Server Fix Packs will have no effect.

MQ_INSTALL_ROOT variable

From WebSphere Application Server 7.0, MQ_INSTALL_ROOT is only used to locate native libraries, and is overridden by any native library path configured on the resource adapter.

Connecting from WebSphere Application Server to IBM MQ

Attention:

1. Any supported version of WebSphere Application Server can use the IBM MQ resource adapter that is bundled with it, to connect to any supported version of IBM MQ.
2. If bindings mode is used, certain libraries in WebSphere Application Server need to match the version of the queue manager to which it is connecting:
   - WebSphere Application Server must be configured to load the native libraries provided with IBM MQ 9.4. See "Configuring the Java Native Interface (JNI) libraries" on page 92 for more information.
   - z/OS On z/OS, you must specify the correct IBM MQ libraries in the WebSphere Application Server STEPLIB concatenation.

     See IBM MQ libraries and the WebSphere Application Server for z/OS STEPLIB for details of the IBM MQ libraries you need.

     If you have libraries for one version of IBM MQ in LINKLIST (LINKLST), you can connect to a different version of IBM MQ by overriding the libraries with STEPLIB.
3. The IBM MQ Resource Adapter version is independent of the native (shared) library versions provided by the queue manager installation.

   For example, WebSphere Application Server 8.5, with an IBM MQ 8.0 Resource Adapter can still manage a bindings connection to an IBM MQ 9.0 queue manager using the IBM MQ 9.0 native libraries.

For more information, see "IBM MQ resource adapter statement of support" on page 422.

The BINDINGS and CLIENT transport types can be used to connect to IBM MQ from any version of WebSphere Application Server. For the BINDINGS transport type, the following restrictions apply:

- IBM MQ must be installed on the same machine as the application server.
- WebSphere Application Server must be configured to load the native libraries provided with IBM MQ.
- z/OS On z/OS, if you want to connect a WebSphere Application Server connection factory to an IBM MQ queue manger in bindings mode, the correct IBM MQ libraries must be specified in the WebSphere Application Server STEPLIB concatenation.

The following table shows the versions of WebSphere Application Server that each version of the IBM MQ resource adapter is supported to run in.

Table 70. Mapping WebSphere Application Server versions to IBM MQ resource adapter versions.	
Version of IBM MQ resource adapter	**Which version of WebSphere Application Server can this version of the resource adapter run in?**
IBM MQ 9.0 and later	The resource adapter can run in: • Any Java EE 7 compliant version of WebSphere Liberty. • WebSphere Application Server traditional 9.0
IBM MQ 8.0	The resource adapter can run in any Java EE 7 compliant version of WebSphere Liberty The IBM MQ 8.0 resource adapter is not supported to run in WebSphere Application Server traditional. The resource adapter already installed in WebSphere Application Server traditional should be used to connect to IBM MQ 8.0 queue managers.

Related concepts

“IBM MQ resource adapter statement of support” on page 422
The IBM MQ resource adapter that you must use for communication between an application and a queue manager depends on whether you are using the Jakarta Messaging 3.0 API or the JMS 2.0 API.

Related information

System Requirements for IBM MQ

Determining the number of TCP/IP connections that are created from WebSphere Application Server to IBM MQ

Using the sharing conversations feature, multiple conversations can share MQI channel instances, this is also known as a TCP/IP connection.

About this task

Applications running inside of WebSphere Application Server 7 and WebSphere Application Server 8, that use IBM MQ messaging provider normal mode, will automatically use this feature. This means that multiple applications running within the same application server instance, that connect to the same IBM MQ queue manager, are able to share the same channel instance.

The number of conversations that can be shared across a single channel instance is determined by the IBM MQ channel property **SHARECNV**. The default value of this property for server connection channels is 10.

By looking at the number of conversations that are created by WebSphere Application Server 7 and WebSphere Application Server 8, you can determine the number of channel instances that are created.

For more information on IBM MQ messaging provider mode, see PROVIDERVERSION normal mode.

Related concepts

Using sharing conversations
In an environment where sharing conversations is permitted, conversations can share an MQI channel instance.

“Sharing a TCP/IP connection in IBM MQ classes for JMS” on page 305

Multiple instances of an MQI channel can be made to share a single TCP/IP connection.

JMS connection factories

Applications running inside of WebSphere Application Server, that use an IBM MQ messaging provider connection factory to create connections and sessions, have active conversations for every JMS connection created from the connection factory, and for every JMS session created from a JMS connection.

One conversation for every JMS connection that has been created from the connection factory

Each JMS connection factory has an associated connection pool, divided into two sections, the free pool and the active pool. Both pools are initially empty.

When an application creates a JMS connection from a connection factory, WebSphere Application Server checks to see if there is a JMS connection in the free pool. If there is, it is moved to the active pool and given to the application. Otherwise, a new JMS connection is created, put in the active pool and returned to the application. The maximum number of connections that can be created from a connection factory is specified by the connection factory connection pool property **Maximum connections**. The default valuue for this property is 10.

After an application has finished with a JMS connection and closed it, the connection is moved from the active pool to the free pool, where it is available for reuse. The connection pool property **Unused timeout** defines how long a JMS connection can stay in the free pool before it is disconnected. The default value for this property is 1800 seconds (30 minutes).

When a JMS connection is first created, a conversation between WebSphere Application Server and IBM MQ starts. The conversation remains active until the connection is closed when value of the **Unused timeout** property for the free pool is exceeded.

One conversation for every JMS session that has been created from a JMS connection

Every JMS connection that is created from a IBM MQ messaging provider connection factory has an associated JMS session pool. There session pools work in the same way as connection pools. The maximum number of JMS Sessions that can be created from a single JMS connection is determined by the connection factory session pool property **Maximum connections**. The default value of this property is 10.

A conversation starts when a JMS session is first created, The conversation remains active until the JMS session is closed because it has remained in the free pool for longer than the value of the **Unused timeout** property for the session pool.

Calculating a value for the SHARECNV property

You can calculate the maximum number of conversations from a single connection factory to IBM MQ by using the following formula:

```
Maximum number of conversations =
    connection Pool Maximum Connections +
   (connection Pool Maximum Connections * Session Pool Maximum Connections)
```

The number of channel instances that will be created to allow this number of conversations to take place can be worked out using the following calculation:

```
Maximum number of channel instances =
    Maximum number of conversations / SHARECNV for the channel being used
```

Any remainder from this calculation can be rounded up.

For a simple connection factory that is using the default value for the connection pool **Maximum connections** and the session pool **Maximum connections** properties, the maximum number of conversations that can exist between WebSphere Application Server and IBM MQ for this connection factory is:

```
Maximum number of conversations =
   connection Pool Maximum Connections +
  (connection Pool Maximum Connections * Session Pool Maximum Connections)
```

For example:

```
        = 10 + (10 * 10)
        = 10 + 100
        = 110
```

If this connection factory is connecting to IBM MQ using a channel that has the **SHARECNV** property set to 10, then the maximum number of channel instances that will be created for this connection factory is:

```
Maximum number of channel instances = Maximum number of conversations / SHARECNV for the
channel being used
```

For example:

```
        = 110 / 10
        = 11 (rounded up to nearest connection)
```

Activation specifications

Message-driven bean applications, that are configured to use an activation specification, have conversations active for activation specification to monitor a JMS destination, and for every server session used to run a message-driven bean instance to process messages.

The following conversations are active for message-driven bean applications that are configured to use an activation specification:

- One conversation for the activation specification to monitor a JMS destination for suitable messages. This conversation is starts as soon as the activation specification starts, and remains active until the activation specification stops.
- One conversation for every server session used to run a message-driven bean instance to process messages.

The activation specification advanced property **Maximum server sessions** specifies the maximum number of server sessions that can be active at any one time for a given activation specification. This property has the default value of 10. Server sessions are created as they are needed, and are closed down if they have been idle for the period of time specified by the activation specification advanced property **Server session pool timeout**. The default value for this property is 300000 milliseconds (5 minutes).

Conversations start when a server session is created, and are stopped either when the activation specification is stopped or when a server session times out.

This means that the maximum number of conversations from a single activation specification to IBM MQ can be calculated using the following formula:

```
Maximum number of conversations = Maximum server sessions + 1
```

The number of channel instances that are created to allow this number of conversations to take place can be found by sing the following calculation:

```
Maximum number of channel instances =
    Maximum number of conversations / SHARECNV for the channel being used
```

Any remainder from this calculation can be rounded up.

For a simple activation specification, that uses the default value for the **Maximum server sessions** property, the maximum number of conversations that can exist between WebSphere Application Server and IBM MQ for this activation specification is calculated as:

```
Maximum number of conversations = Maximum server sessions + 1
```

For example:

```
    = 10 + 1
    = 11
```

If this activation specification is connecting to IBM MQ using a channel that has the **SHARECNV** property set to 10, then the number of channel instances that are created is calculated as:

```
Maximum number of channel instances =
    Maximum number of conversations / SHARECNV for the channel being used
```

For example:

```
    = 11 / 10
    = 2 (rounded up to nearest connection)
```

Listener ports running in Application Server Facilities (ASF) mode

Listener ports running in ASF mode used by message-driven bean applications create conversations for each server session. One monitors a destination for suitable messages and another runs a message-driven bean instance to process messages. The number of conversations for each listener port can be calculated from a maximum number of sessions.

By default, listener ports will run in ASF mode as part of the 1.1 specification that defines the mechanism which application servers should use to detect messages and deliver them to message-driven beans for processing. Message-driven bean applications that are set up to use listener ports in this default mode of operation create conversations:

One conversation for the listener port to monitor a destination for suitable messages

Listener ports are configured to use a JMS connection factory. When a listener port starts, a request is made for a JMS connection from the connection factory free pool. The connection is returned to the free pool when the listener port is stopped. For more information about how the connection pool is used, and how this affects the number of conversations to IBM MQ, see “JMS connection factories” on page 485.

One conversation for every server session used to run a message-driven bean instance to process messages

The listener port property **Maximum sessions** specifies the maximum number of server sessions that can be active at any one time for a given listener port. This property has the default value of 10. Server sessions are created as they are needed, and make use of JMS sessions taken from the session pool associated with the JMS connection that the listener port is using.

If a server session has been idle for the period of time specified by the Message Listener Service custom property **SERVER.SESSION.POOL.UNUSED.TIMEOUT**, the session is closed and the JMS session used is returned to the session pool free pool. The JMS session will remain in the session pool free pool until it is needed, or it is closed because it has been idle in the free pool for longer than the value of the session pool's **Unused timeout** property.

For more information about how the session pool is used, and how the conversations between WebSphere Application Server and IBM MQ are managed, see "JMS connection factories" on page 485.

For more information about the Message Listener Service custom property **SERVER.SESSION.POOL.UNUSED.TIMEOUT**, see Monitoring server session pools for listener ports in the WebSphere Application Server product documentation.

Calculating the maximum number of conversations from a single listener port to IBM MQ

You can calculate the maximum number of conversations from a single listener port to IBM MQ by using the following formula:

```
Maximum number of conversations = Maximum sessions + 1
```

The number of channel instances that will be created to allow this number of conversations to take place can be worked out using the following calculation:

```
     Maximum number of channel instances =
        Maximum number of conversations / SHARECNV for the channel being used
```

Any remainder from this calculation can be rounded up.

For a simple listener port that is using the default value for the **Maximum sessions** property, the maximum number of conversations that can exist between WebSphere Application Server and IBM MQ for this listener port is calculated as:

```
Maximum number of conversations = Maximum sessions + 1
```

For example:

```
    = 10 + 1
    = 11
```

If this listener port is connecting to IBM MQ using a channel that has the **SHARECNV** property set to 10, then the number of channel instances that will be created is calculated as:

```
Maximum number of channel instances =
    Maximum number of conversations / SHARECNV for the channel being used
```

For example:

```
    = 11 / 10
    = 2 (rounded up to nearest connection)
```

Listener ports running in non Application Server Facilities (non-ASF) mode

Listener ports running in non-ASF mode can be configured to monitor the queue destination and the topic destination using Server Sessions. Server sessions can have multiple conversations, the maximum number of which can be calculated in each case.

Listener ports can be configured to run in non-ASF mode which changes the way the listener ports monitor JMS destinations. Message-driven bean applications, using listener ports in non-ASF mode of operation, create a conversation for every server session used to run a message-driven bean instance to process messages. The listener port property **maximum sessions** specifies the maximum number of Server Sessions that can be active at any one time for a given listener port. The default value for this property is 10.

When running in non-ASF mode, a listener port monitoring a queue destination will automatically create the number of Server Sessions specified by the listener port property **Maximum sessions**. All of these Server Sessions make use of JMS Sessions taken from the session pool associated with the JMS Connection that the listener port is using, and continually monitor a JMS Destination for suitable messages.

If the listener port is configured to monitor a topic destination, the value of **Maximum sessions** is ignored and a single Server Session is used.

The Server Sessions used by a listener port running in non-ASF mode remain active until the listener port is stopped, at which point the JMS Sessions that were used are returned to the session pool Free Pool for the JMS Connection that the listener port was using.

For more information about how the session pool is used, and how the conversations between WebSphere Application Server and IBM MQ are managed, see "JMS connection factories" on page 485.

For more information about ASF and non-ASF mode of operation with WebSphere Application Server, and how to configure Listener Ports to use non-ASF mode, see Message processing in ASF mode and non-ASF mode.

Calculating the maximum number of conversations while monitoring a queue destination

The maximum number of conversations from a single listener port, running in non-ASF mode and monitoring a queue destination to IBM MQ can be calculated using the following formula:

```
Maximum number of conversations = Maximum sessions
```

The number of channel instances that will be created to allow this number of conversations to take place can be found by using the following calculation:

```
Maximum number of channel instances =
   Maximum number of conversations / SHARECNV for the channel being used
```

Any remainder from this calculation can be rounded up.

For a simple listener port running in non-ASF mode that is using the default value for the **Maximum sessions** property and monitoring a queue destination, the maximum number of conversations that can exist between WebSphere Application Server and IBM MQ for this listener port is:

```
Maximum number of conversations = Maximum sessions
```

For example:

```
= 10
```

If this listener port is connecting to IBM MQ using a channel that has the **SHARECNV** property set to 10, then the number of channel instances that are created is calculated as:

```
Maximum number of channel instances =
   Maximum number of conversations / SHARECNV for the channel being used
```

For example:

```
= 10 / 10
= 1
```

Calculating the maximum number of conversations while monitoring a topic destination

For a listener port running in non-ASF mode and configured to monitor a topic destination, the number of conversations from the listener port to IBM MQ is:

```
Maximum number of conversations = 1
```

The number of channel instances that will be created to allow this number of conversations to take place can be found by using the following calculation:

```
Maximum number of channel instances =
   Maximum number of conversations / SHARECNV for the channel being used
```

Any remainder from this calculation can be rounded up.

For a simple listener port running in non-ASF mode that is using the default value for the **Maximum sessions** property and monitoring a topic destination, the maximum number of conversations that can exist between WebSphere Application Server and IBM MQ for this listener port is:

```
Maximum number of conversations = Maximum sessions
```

For example:

```
= 10
```

If this listener port is connecting to IBM MQ using a channel that has the **SHARECNV** property set to 10, then the number of channel instances that are created is calculated as:

```
Maximum number of channel instances =
   Maximum number of conversations / SHARECNV for the channel being used
```

For example:

```
= 10 / 10
= 1
```

Configuring authentication aliases to secure WebSphere Application Server connection to IBM MQ

Authentication aliases map to a user name and password combination that can be used to secure WebSphere Application Server connection to IBM MQ. You can configure a connection factory with an authentication alias.

Using authentication aliases with enterprise applications

When an enterprise application running inside of WebSphere Application Server attempts to create a JMS connection to IBM MQ, the application looks up an IBM MQ messaging provider connection factory definition from the Java Naming Directory Interface (JNDI) repository of the application server.

When the IBM MQ messaging provider connection factory definition is located from within the JNDI repository of the application server, one of the following methods is called:

- `ConnectionFactory.createConnection()`
- `ConnectionFactory.createConnection(String username, String password)`

If the connection factory has been configured with a J2C authentication alias defined, then the user name and password in the authentication alias can be flowed down to IBM MQ when the connection factory is used to create a connection.

Connection factories and authentication aliases

IBM MQ messaging provider connection factories contain information on how to connect to IBM MQ queue managers. Enterprise applications running inside of WebSphere Application Server can use the connection factories to create JMS connections to IBM MQ.

WebSphere Application Server stores connection factories definitions in a repository that can be accessed using the JNDI. When a connection factory is created, the connection factory is given a JNDI name to uniquely identify it at the application server scope (either the Cell, Node or Server scope) at which it has been defined.

For example, an IBM MQ messaging provider connection factory defined at the WebSphere Application Server Cell scope contains information on how to connect to the queue manager (`myQM`) using the BINDINGS transport. This connection factory is given the JNDI name `jms/myCF` to uniquely identify it.

Connection factories can also be configured to use an authentication alias. Authentication aliases map to a user name and password combination. Depending on how the connection factory is used, the user name and password in the authentication alias might, or might not, be flowed down to IBM MQ when the JMS connection is created.

Important: Prior to IBM MQ 8.0, the default IBM MQ Object Authority Manager (OAM) performed an authorization check, only to ensure that the user name passed down to IBM MQ, when a connection is made, had the authority to access the queue manager.

No checks were made to validate the password that was specified. In order to perform an authentication check, and validate that the user identifier and password match, you needed to write an IBM MQ channel security exit. Details on how to do this can be found in "Channel security exit programs" on page 933.

From IBM MQ 8.0, the queue manager checks the password in addition to the user name.

Using the connection factory

The following topics contain information about using the connection factory using direct and indirect look ups:

- "Using the connection factory through a direct lookup" on page 494
- "Using the connection factory through an indirect lookup" on page 495

Using the CLIENT transport

Connection factories that are configured to use the CLIENT transport must specify which IBM MQ server connection channel (SVRCONN) they are going to use to connect to the queue manager.

If the IBM MQ channel agent user identifier (MCAUSER) property remains blank for the channel that the connection factory has been configured to use, then the connection factory can be used with either a direct look up, or indirect look up.

If the MCAUSER property is set to a user identifier, this user identifier is passed down to IBM MQ when the connection factory is used to create a connection to IBM MQ, regardless of whether the enterprise application is using a direct or indirect look up.

Summary tables

The following tables summarize what user identifiers are flowed down to IBM MQ when the BINDINGS transport, and the CLIENT transport, respectively are used:

Table 71. BINDINGS mode

Configuration	**Application calls `ConnectionFactory.createConnection()`**	**Application calls `ConnectionFactory.createConnection(String username, String password)`**
Application's deployment descriptor does not contain a Resource Reference for the connection factory	The user identifier for the application server process is flowed down to IBM MQ.	The user identifier and password that were passed into the `ConnectionFactory.createConnection(String username, String password)` method are flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory and the **`res-auth`** property is set to "Application"	The user identifier for the application server process is flowed down to IBM MQ.	The user identifier and password that were passed into the `ConnectionFactory.createConnection(String username, String password)` method are flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory and the **`res-auth`** property is set to "Container"	The user identifier and password specified in the authentication alias for the connection factory are flowed down to IBM MQ.	The user identifier and password specified in the authentication alias for the connection factory are flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property set to "Container" and the application has been configured with an authentication alias	The user identifier and password specified in the authentication alias that the application has been configured to use are flowed down to IBM MQ.	The user identifier and password specified in the authentication alias that the application has been configured to use are flowed down to IBM MQ.

Table 72. CLIENT mode

Configuration	**Application calls `ConnectionFactory.createConnection()`**	**Application calls `ConnectionFactory.createConnection(String username, String password)`**
Application's deployment descriptor does not contain a Resource Reference for the connection factory and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property unset	The user identifier for the application server process is flowed down to IBM MQ.	The user identifier and password that were passed into the `ConnectionFactory.createConnection(String username, String password)` method are flowed down to IBM MQ.

Table 72. CLIENT mode (continued)

Configuration	Application calls **`ConnectionFactory.createConnection()`**	Application calls **`ConnectionFactory.createConnection(String username, String password)`**
Application's deployment descriptor does not contain a Resource Reference for the connection factory and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property set to a user identifier	The user identifier specified by the MCAUSER property on the IBM MQ channel the connection factory is configured to use is flowed down to IBM MQ.	The user identifier specified by the MCAUSER property on the IBM MQ channel the connection factory is configured to use is flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property is set to *Application* and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property unset	The user identifier for the application server process is flowed down to IBM MQ.	The user identifier and password that were passed into the `ConnectionFactory.createConnection(String username, String password)` method are flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property is set to *Application* and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property set to a user identifier	The user identifier specified by the MCAUSER property on the IBM MQ channel which the connection factory is configured to use is flowed down to IBM MQ.	The user identifier specified by the MCAUSER property on the IBM MQ channel which the connection factory is configured to use is flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property is set to"*Container* and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property unset	The user identifier and password specified in the authentication alias for the connection factory are flowed down to IBM MQ.	The user identifier and password specified in the authentication alias for the connection factory are flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property is set to"*Container* and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property set to a user identifier	The user identifier specified by the MCAUSER property on the IBM MQ channel which the connection factory is configured to use is flowed down to IBM MQ.	The user identifier specified by the MCAUSER property on the IBM MQ channel which the connection factory is configured to use is flowed down to IBM MQ.

Table 72. CLIENT mode (continued)		
Configuration	**Application calls `ConnectionFactory.createConnection()`**	**Application calls `ConnectionFactory.createConnection(String username, String password)`**
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property is set to"*Container* and the application has been configured with an authentication alias and the connection factory is configured to use an IBM MQ channel that has the MCAUSER property unset	The user identifier and password specified in the authentication alias that the application has been configured to use are flowed down to IBM MQ.	The user identifier and password specified in the authentication alias that the application has been configured to use are flowed down to IBM MQ.
Application's deployment descriptor contains a Resource Reference for the connection factory which has the **`res-auth`** property is set to *Container* and the application has been configured with an authentication alias and the connection factory is configured to use an IBM MQ channel that has the MCAUSER set to a user identifier	The user identifier specified by the MCAUSER property on the IBM MQ channel which the connection factory is configured to use is flowed down to IBM MQ.	The user identifier specified by the MCAUSER property on the IBM MQ channel which the connection factory is configured to use is flowed down to IBM MQ.

Using the connection factory through a direct lookup

After an IBM MQ messaging provider connection factory has been defined, an enterprise application can look up the connection factory definition and use it to create a JMS connection to an IBM MQ queue manager. This can be done through a direct look up.

To use a direct lookup, an enterprise application connects to the JNDI repository of the application server, by making the following method call:

```
InitialContext ctx = new InitialContext();
```

Once it has connected to the JNDI repository, the enterprise application then identifies the connection factory definition using the JNDI name of the connection factory, as follows:

```
ConnectionFactory cf = (ConnectionFactory) ctx.lookup("jms/myCF");
```

Notes:

- Your application developer needs to know the JNDI name of the required connection factory when the enterprise application is being developed. Because the JNDI name is hard coded inside the application, if the JNDI name changes, you need to re-write and re-deploy the application.
- When a connection factory definition is used in this way, the user name and password specified in the authentication alias (that the connection factory has been configured to use) are not flowed down to IBM MQ. This is to prevent unauthorized applications from identifying the connection factory, and being able to use it to connect to secure IBM MQ systems.

The user name and password that are flowed down to IBM MQ depends on the method that is used to create the JMS connection from the connection factory.

If an application creates a JMS connection using the method:

```
ConnectionFactory.createConnection()
```

the default user identity is passed down to IBM MQ. This is the user name and password that started the application server where the enterprise application is running.

Alternatively, an application can create a JMS connection is by calling the method:

```
ConnectionFactory.createConnection(String username, String password)
```

If an application has performed a direct look up of a connection factory, and then called this method, the user name and password that were passed into the `createConnection()` method are flowed down to IBM MQ.

Important: Prior to IBM MQ 8.0, IBM MQ processed an authorization check, only to make sure that the user name that had been flowed down, had the authority to access the queue manager.

No checks were made on the password. In order to perform an authentication check, and validate that the user name and password were valid, an IBM MQ channel security exit must be written. Details on how to do this can be found in “Channel security exit programs” on page 933.

From IBM MQ 8.0, the queue manager checks the password in addition to the user name.

Using the connection factory through an indirect lookup

When you are writing an enterprise application, if the JNDI name of the connection factory is unknown, or if the application is to be installed onto different application servers using a different connection factory, with a different JNDI name (depending on what application server it is installed onto), then the connection factory can be looked up using a resource reference. This can be done through an indirect lookup.

Example

Rather than directly looking up the connection factory using `jms/myCF`, an enterprise application contains a resource reference has the local JNDI name of: `jms/myResourceReferenceCF`.

To use this JNDI name, the application connects to the JNDI repository of the application server, in the same way as if the application is performing a direct look up:

```
InitialContext ctx = new InitialContext();
```

Rather than identifying `jms/myCF` directly, the application now identifies the JNDI name of the resource reference:

```
ConnectionFactory cf = (ConnectionFactory) ctx.lookup("java:comp/env/jms/
myResourceReferenceCF");
```

You need the `java:comp/env` prefix for the local JNDI name, to tell the application server that the enterprise application is performing an indirect look up.

When the application is deployed, the user maps the JNDI name of the resource reference `jms/myResourceReferenceCF` to the JNDI name of the connection factory that the application has already created: `jms/myCF`.

When the application is run, it looks up a JMS connection factory using the local JNDI name, which the application server maps onto: `jms/myCF`. This connection factory is then used by the application to create a connection to IBM MQ.

Authentication aliases and indirect lookups

A resource reference also allows additional properties to be defined, that alter the behavior of the provided connection factory. One of the properties of a resource reference is **res-auth**. The value of this property specifies whether the enterprise application should use the authentication alias of the connection factory that the resource reference maps to when creating a connection to IBM MQ (if an authentication alias has been defined), or if the application is specifying its own user name and password.

The default value of this property is *Application*. This means that the user name and password that are flowed down to the queue manager, when a JMS connection is created, is determined by the application itself. The authentication alias of the connection factory that the resource reference maps to is not used.

Applications can create JMS connections using one of the following methods:

- `ConnectionFactory.createConnection()`
- `ConnectionFactory.createConnection(String username, String password)`

If an application uses `ConnectionFactory.createConnection()`, and **res-auth** is set to *Application*, the default user identity is flowed down to IBM MQ. This is the user name and password that started the application server where the enterprise application is running.

If an application uses `ConnectionFactory.createConnection(String username, String password)`, and **res-auth** is set to *Application*, the user name and password passed in to the method are sent down to IBM MQ.

In order to use the authentication alias defined on the connection factory that the resource reference maps to when creating a connection, you need to set the **res-auth** property to the value *Container*. When an application creates a JMS connection, the authentication alias details are used, even if the `createConnection` call specifies a user name and password.

Overriding the authentication alias when using an indirect lookup

If an application uses a resource reference that has the **res-auth** property set to *Container*, you can override the authentication alias that is used when JMS connections are created.

To override the authentication alias, the resource reference needs to include an extra property called **authDataAlias**, that maps to an existing authentication alias that has already been created in the application server environment into which the application will be deployed. You can specify this property on any resource references that are created using the Rational® tooling provided by IBM.

Using this method, you can use a different authentication alias when using a JMS connection factory that has been looked up indirectly. If the authentication alias specified does not exist, then a new one can be specified after the enterprise application has been installed. For more information, see *Resource references* in the WebSphere Application Server product documentation.

Related information for WebSphere Application Server 8.5.5

Resource references

Related information for WebSphere Application Server 8.0

Resource references

Related information for WebSphere Application Server 7.0

Resource references

Workload balancing for message driven beans when using WebSphere Application Server clusters

When using message driven bean applications deployed in a WebSphere Application Server 7.0 and WebSphere Application Server 8.0 cluster, and configured to run in IBM MQ messaging provider normal mode, one of the cluster members processes the majority of the messages. You can balance the workload of cluster members in order to distribute the processing of messages across more than one cluster member.

IBM MQ includes a feature called **Asynchronous consume**, which allows applications to consume messages asynchronously from a queue using APIs called **MQCB** and **MQCTL**.

Message driven bean applications running inside of WebSphere Application Server 7.0 and WebSphere Application Server 8.0, that use IBM MQ messaging provider normal mode will automatically make use of this feature. When the applications start up, they will set up an asynchronous consumer on the JMS destination that they have been configured to monitor by calling **MQCB**. The **MQCTL** API is then called to indicate that the application is ready to receive messages from the JMS destination.

When message driven bean applications have been deployed into a WebSphere Application Server cluster, each cluster member will set up an asynchronous consumer for the JMS destination that the message-driven bean is monitoring for messages. The IBM MQ queue manager that hosts the JMS destination is then responsible for notifying the cluster member when there is a suitable message on the JMS destination for it to process.

When WebSphere Application Server is connecting to an IBM MQ queue manager, messages that arrive on a JMS destination will be distributed more evenly to all of the asynchronous consumers that have been registered on that JMS destination. For message-driven bean applications deployed inside of a WebSphere Application Server 7.0 and WebSphere Application Server 8.0 cluster, this means that the messages will be distributed more evenly between cluster members.

Related tasks

Configuring the JMS **PROVIDERVERSION** property

Using the IBM MQ Headers package

The IBM MQ Headers package provides a set of helper interfaces and classes that you can use to manipulate the IBM MQ headers of a message. Typically, you use the IBM MQ Headers package because you want to perform administrative services by using the command server (by using Programmable Command Format (PCF) messages).

About this task

The IBM MQ Headers package is located in the `com.ibm.mq.headers` and `com.ibm.mq.headers.pcf` packages. You can use this facility for both of the two alternative APIs that IBM MQ provides for use in Java applications:

- IBM MQ classes for Java (also referred to as IBM MQ Base Java).
- IBM MQ classes for Java Message Service (IBM MQ classes for JMS, also referred to as IBM MQ JMS).

IBM MQ Base Java applications typically manipulate MQMessage objects, and the Headers support classes can directly interact with these objects, since they natively understand the IBM MQ Base Java interfaces.

In IBM MQ JMS, the payload for a message is typically a String or a byte array object, which can be manipulated with DataInput and DataOutput streams. The IBM MQ Headers package can be used to interact with these data streams and is suitable for manipulating any MQ messages that are sent and received by IBM MQ JMS applications.

Therefore, although the IBM MQ Headers package contains references to the IBM MQ Base Java package, it is also intended for use within IBM MQ JMS applications and is suitable for use within Java Platform, Enterprise Edition (Java EE) environments.

A typical way in which you might use the IBM MQ Headers package is to manipulate administration messages in Programmable Command Format (PCF), for example for any of the following reasons:

- To access details about an IBM MQ resource.
- To monitor the depth of a queue.
- To inhibit access to a queue.

By using PCF messages with the IBM MQ JMS API, this kind of administration of application-centric resources can be performed from within Java EE applications without having to resort to using the IBM MQ Base Java API.

Procedure

- To use the IBM MQ Headers package to manipulate message headers for IBM MQ classes for Java, see "Using with IBM MQ classes for Java" on page 498.
- To use the IBM MQ Headers package to manipulate message headers for IBM MQ classes for JMS, see "Using with IBM MQ classes for JMS" on page 498.

Using with IBM MQ classes for Java

IBM MQ classes for Java applications typically manipulate MQMessage objects, and the Headers support classes can directly interact with these objects, since they natively understand the IBM MQ classes for Java interfaces.

About this task

IBM MQ provides some sample applications that demonstrate how to use the IBM MQ Headers package with the IBM MQ Base Java API (IBM MQ classes for Java).

The samples show two things:

- How to create a PCF message to perform an administrative action and parse the response message.
- How to send this PCF message using the IBM MQ classes for Java.

Depending on which platform you are using, these samples are installed under the `pcf` directory in the `samples` or `tools` directory of your IBM MQ installation (see "Installation directories for IBM MQ classes for Java" on page 342).

Procedure

1. Create a PCF message to perform an administrative action and parse the response message.
2. Send this PCF message using the IBM MQ classes for Java.

Related concepts

"Handling IBM MQ message headers with IBM MQ classes for Java" on page 368
Java classes are provided representing different types of message header. Two helper classes are also provided.

"Handling PCF messages with IBM MQ classes for Java" on page 373
Java classes are provided to create and parse PCF-structured messages, and to facilitate sending PCF requests and collecting PCF responses.

Using with IBM MQ classes for JMS

To use the IBM MQ Headers with the IBM MQ classes for JMS, you carry out the same essential steps as for IBM MQ classes for Java. The PCF message can be created and the response parsed in exactly the same way by using the IBM MQ Headers package and the same sample code as for IBM MQ classes for Java.

About this task

To send a PCF message using the IBM MQ API, the message payload must be written into a JMS Bytes Message, and sent using the standard JMS APIs. The only consideration is that the message must not contain a JMS RFH2 or any other headers with specific values in the MQMD.

To send a PCF message, complete the following steps. The way in which the PCF message is created, and information is extracted from the response message is the same as for IBM MQ classes for Java (see “Using with IBM MQ classes for Java” on page 498).

Procedure

1. Create a JMS Queue Destination that represents the SYSTEM.ADMIN.COMMAND.QUEUE.

   IBM MQ JMS applications send the PCF messages to the SYSTEM.ADMIN.COMMAND.QUEUE, and need access to a JMS Destination object that represents this queue. The Destination must have the following properties set:

   ```
   WMQ_MQMD_WRITE_ENABLED = YES
   WMQ_MESSAGE_BODY = MQ
   ```

   If you are using WebSphere Application Server, you must define these properties as custom properties on the Destination.

   To create the destination programmatically from within an application, use the following code:

   ```
   Queue q1 = session.createQueue("SYSTEM.ADMIN.COMMAND.QUEUE");
   ((MQQueue) q1).setIntProperty(WMQConstants.WMQ_MESSAGE_BODY,
       WMQConstants.WMQ_MESSAGE_BODY_MQ);
   ((MQQueue) q1).setMQMDWriteEnabled(true);
   ```

2. Convert a PCF message into a JMS Bytes message containing the correct MQMD values.

   A JMS Bytes message needs to be created, and the PCF Message written to it. A response queue needs to be created, but this needs to have no specific settings.

   The following sample code snippet shows how to create a JMS Bytes Message and write a com.ibm.mq.headers,pcf.PCFMessage object into it. The PCFMessage object (pcfCmd) has previously been built using the IBM MQ Headers package. (Note the package to load the PCFMessage is com.ibm.mq.headers.pcf.PCFMessage).

   ```
    // create the JMS Bytes Message
   final BytesMessage msg = session.createBytesMessage();

   // Create the wrapping streams to put the bytes into the message payload
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
   DataOutput dataOutput = new DataOutputStream(baos);

   // Set the JMSReplyTo so the answer comes back
   msg.setJMSReplyTo(new MQQueue("adminResp"));

   // write the pcf into the stream
   pcfCmd.write(dataOutput);
   baos.flush();
   msg.writeBytes(baos.toByteArray());

   // we have taken control of the MD, so need to set all
   // flags in the MD that we require - main one is the format
   msg.setJMSPriority(4);
   msg.setIntProperty(WMQConstants.JMS_IBM_MQMD_PERSISTENCE,
     CMQC.MQPER_NOT_PERSISTENT);
   msg.setIntProperty(WMQConstants.JMS_IBM_MQMD_EXPIRY, 300);
   msg.setIntProperty(WMQConstants.JMS_IBM_MQMD_REPORT,
       CMQC.MQRO_PASS_CORREL_ID);
   msg.setStringProperty(WMQConstants.JMS_IBM_MQMD_FORMAT, "MQADMIN");

   // and send the message
   sender.send(msg);
   ```

3. Send the message, and receive the response using the standard JMS APIs.
4. Convert the response message into a PCF message for processing.

   To retrieve the response message and process it as a PCF message, use the following code:

   ```
    // Get the message back
   BytesMessage msg = (BytesMessage) consumer.receive();

   // get the size of the bytes message & read into an array
   int bodySize = (int) msg.getBodyLength();
   ```

```
byte[] data = new byte[bodySize];
msg.readBytes(data);

// Read into Stream and DataInput Stream
ByteArrayInputStream bais = new ByteArrayInputStream(data);
DataInput dataInput = new DataInputStream(bais);

// Pass to PCF Message to process
PCFMessage response = new PCFMessage(dataInput);
```

Related concepts

"JMS messages" on page 138
JMS messages are composed of a header, properties, and a body. JMS defines five types of message body.

IBM i Setting up IBM MQ on IBM i with Java and JMS

This collection of topics gives an overview of how you set up and test IBM MQ with Java and JMS on IBM i using CL commands or the qshell environment.

Note:

- From IBM MQ 8.0, `ldap.jar`, `jndi.jar` and `jta.jar` are part of the JDK.
- JMS 3.0 From IBM MQ 9.3.0, Jakarta Messaging 3.0 is supported for developing new applications. IBM MQ 9.3.0 and later continue to support JMS 2.0 for existing applications. It is not supported to use both the Jakarta Messaging 3.0 API and the JMS 2.0 API in the same application. For more information, see Using IBM MQ classes for JMS/Jakarta Messaging.

Using CL commands

The CLASSPATH that you set, is for testing with MQ base Java, JMS with JNDI, and JMS without JNDI.

If you do not use a `.profile` file under your `/home/Userprofile` directory, you will need to set the system level environment variables below. You can check to see if they are set using the **WRKENVVAR** command.

1. To view the environment variables for the entire system issue the command: **WRKENVVAR LEVEL(*SYS)**
2. To view the environment variables specific to your job issue the command : **WRKENVVAR LEVEL(*JOB)**
3. If the CLASSPATH is not set, issue the following command:

JMS 3.0
```
ADDENVVAR ENVVAR(CLASSPATH)
       VALUE('.:/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar
       :/QIBM/ProdData/mqm/java/lib/connector.jar:/QIBM/ProdData/mqm/java/lib
       :/QIBM/ProdData/mqm/java/samples/base
       :/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jakarta.client.jar
       :/QIBM/ProdData/mqm/java/lib/jms.jar
       :/QIBM/ProdData/mqm/java/lib/providerutil.jar
       :/QIBM/ProdData/mqm/java/lib/fscontext.jar:') LEVEL(*SYS)
```

JMS 2.0
```
ADDENVVAR ENVVAR(CLASSPATH)
       VALUE('.:/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar
       :/QIBM/ProdData/mqm/java/lib/connector.jar:/QIBM/ProdData/mqm/java/lib
       :/QIBM/ProdData/mqm/java/samples/base
       :/QIBM/ProdData/mqm/java/lib/com.ibm.mq.allclient.jar
       :/QIBM/ProdData/mqm/java/lib/jms.jar
       :/QIBM/ProdData/mqm/java/lib/providerutil.jar
       :/QIBM/ProdData/mqm/java/lib/fscontext.jar:') LEVEL(*SYS)
```

4. If QIBM_MULTI_THREADED is not set, issue the following command:

```
ADDENVVAR ENVVAR(QIBM_MULTI_THREADED) VALUE('Y') LEVEL(*SYS)
```

5. If QIBM_USE_DESCRIPTOR_STDIO is not set, issue the following command:

```
ADDENVVAR ENVVAR(QIBM_USE_DESCRIPTOR_STDIO) VALUE('I') LEVEL(*SYS)
```

6. If QSH_REDIRECTION_TEXTDATA is not set , issue the following command:

```
ADDENVVAR ENVVAR(QSH_REDIRECTION_TEXTDATA) VALUE('Y') LEVEL(*SYS)
```

Using the qshell environment

If you use the QSHELL environment, you can set up a `.profile` in your `/home/Userprofile` directory. For more information reference the Qshell Interpreter (qsh) documentation.

Specify the following in the `.profile`. Note that the CLASSPATH statement must be on a single line, or separated onto different lines using the \ character as shown.

JMS 3.0

```
CLASSPATH=.:/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar: \
/QIBM/ProdData/mqm/java/lib/connector.jar: \
/QIBM/ProdData/mqm/java/lib: \
/QIBM/ProdData/mqm/java/samples/base: \
/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jakarta.client.jar: \
/QIBM/ProdData/mqm/java/lib/jms.jar: \
/QIBM/ProdData/mqm/java/lib/providerutil.jar: \
/QIBM/ProdData/mqm/java/lib/fscontext.jar:
HOME=/home/XXXXX
LOGNAME=XXXXX
PATH=/usr/bin:
QIBM_MULTI_THREADED=Y QIBM_USE_DESCRIPTOR_STDIO=I
QSH_REDIRECTION_TEXTDATA=Y
TERMINAL_TYPE=5250
```

JMS 2.0

```
CLASSPATH=.:/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar: \
/QIBM/ProdData/mqm/java/lib/connector.jar: \
/QIBM/ProdData/mqm/java/lib: \
/QIBM/ProdData/mqm/java/samples/base: \
/QIBM/ProdData/mqm/java/lib/com.ibm.mq.allclient.jar: \
/QIBM/ProdData/mqm/java/lib/jms.jar: \
/QIBM/ProdData/mqm/java/lib/providerutil.jar: \
/QIBM/ProdData/mqm/java/lib/fscontext.jar:
HOME=/home/XXXXX
LOGNAME=XXXXX
PATH=/usr/bin:
QIBM_MULTI_THREADED=Y QIBM_USE_DESCRIPTOR_STDIO=I
QSH_REDIRECTION_TEXTDATA=Y
TERMINAL_TYPE=5250
```

Ensure that the QMQMJAVA library is in the library list by issuing the command **DSPLIBL**.

If the QMQMJAVA library is not in the list, add it using the following command: **ADDLIBLE LIB(QMQMJAVA)**

IBM i Testing IBM MQ on IBM i with Java

How you test IBM MQ with Java using the MQIVP sample program.

Testing IBM MQ base Java

Carry out the following procedure:

1. Verify that the queue manager is started, and that the state of the queue manager is ACTIVE, by issuing the following command:

```
WRKMQM MQMNAME(QMGRNAME)
```

2. Verify that the JAVA.CHANNEL server connection channel has been created by issuing the following command:

```
WRKMQMCHL CHLNAME(JAVA.CHANNEL) CHLTYPE(*SVRCN) MQMNAME(QMGRNAME)
```

   a. If the JAVA.CHANNEL does not exist, issue the following command:

```
CRTMQMCHL CHLNAME(JAVA.CHANNEL) CHLTYPE(*SVRCN) MQMNAME(QMGRNAME)
```

3. Verify the queue manager listener is running for port 1414 or whichever port you are using, by issuing the **WRKMQMLSR** command.

   a. If no listener has been started for the queue manager, issue the following command:

```
STRMQMLSR PORT(xxxx) MQMNAME(QMGRNAME)
```

Running the MQIVP sample test program

1. Start the qshell, from the command line by issuing the command STRQSH
2. Verify that the correct CLASSPATH is set by issuing the **export** command, and then issue the **cd** command as follows:

```
cd /qibm/proddata/mqm/java/samples/wmqjava/samples
```

3. Run the **java** program by issuing the following command:

```
java MQIVP
```

You can press the ENTER key when prompted for:

- Type of connection
- IP address
- Queue manager name

to use the default values. This verifies the product bindings, which can be found in the QMQMJAVA library.

You receive output similar to the following example. Note that the copyright statement depends upon the version of the product that you are using.

```
> java MQIVP
MQSeries for Java Installation Verification Program
5724-H72 (C) Copyright IBM Corp. 2011, 2024. All Rights Reserved.
============================================================

Please enter the IP address of the MQ server :>
Please enter the queue manager name :>
Attaching Java program to QIBM/ProdData/mqm/java/lib/connector.JAR.
Success: Connected to queue manager.
Success: Opened SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Put a message to SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Got a message from SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Closed SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Disconnected from queue manager


Tests complete -
SUCCESS: This MQ Transport is functioning correctly.
Press Enter to continue ...>
$
```

Testing IBM MQ Java client connection

You must specify the:

- Connection type
- IP address
- Port
- Server connection channel
- Queue manager

You receive output similar to the following example. Note that the copyright statement depends upon the version of the product that you are using.

```
> java MQIVP
MQSeries for Java Installation Verification Program
5724-H72 (C) Copyright IBM Corp. 2011, 2024. All Rights Reserved.
============================================================

Please enter the IP address of the MQ server :> x.xx.xx.xx
Please enter the port to connect to : (1414)> 1470
Please enter the server connection channel name :> JAVA.CHANNEL
Please enter the queue manager name :> KAREN01
Success: Connected to queue manager.
Success: Opened SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Put a message to SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Got a message from SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Closed SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Disconnected from queue manager


Tests complete -
SUCCESS: This MQ Transport is functioning correctly.
Press Enter to continue ...>
$
```

IBM i Testing IBM MQ on IBM i with JMS

How you test IBM MQ with JMS with and without JNDI

Testing JMS without JNDI using the IVTRun sample

Carry out the following procedure:

1. Verify that the queue manager is started, and that the state of the queue manager is ACTIVE, by issuing the following command:

   ```
   WRKMQM MQMNAME(QMGRNAME)
   ```

2. Start the qshell, from the command line, by issuing the **STRQSH** command.
3. Use the **cd** command to change directory as follows:

   ```
   cd /qibm/proddata/mqm/java/bin
   ```

4. Run the script file:

   ```
   IVTRun -nojndi [-m qmgrname]
   ```

You receive output similar to the following example. Note that the copyright statements depend upon the versions of the products that you are using:

```
 IVTRun -nojndi -m ELCRTP19

Attaching Java program to
/QIBM/ProdData/mqm/java/lib/com.ibm.mqjms.JAR.
```

```
Attaching Java program to
/QIBM/ProdData/mqm/java/lib/jms.JAR.

5724-H72, 5724-B41, 5655-F10 (c) Copyright IBM Corp. 2011, 2024.
All Rights Reserved.
WebSphere MQ classes for Java Message Service 5.300
Installation Verification Test


Creating a QueueConnectionFactory
Creating a Connection
Creating a Session
Creating a Queue
Creating a QueueSender
Creating a QueueReceiver
Creating a TextMessage
Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE
Reading the message back again

Got message:
JMS Message class: jms_text
JMSType: null
JMSDeliveryMode: 2
JMSExpiration: 0
JMSPriority: 4
JMSMessageID: ID:c1d4d840c5d3c3d9e3d7f1f9404040403ccf041f0000c012
JMSTimestamp: 1020273404500
JMSCorrelationID:null
JMSDestination: queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
JMSReplyTo: null
JMSRedelivered: false
JMS_IBM_PutDate:20040326
JMSXAppID:QP0ZSPWT STANLEY 170302
JMS_IBM_Format:MQSTR
JMS_IBM_PutApplType:8
JMS_IBM_MsgType:8
JMSXUserID:STANLEY
JMS_IBM_PutTime:13441354
JMSXDeliveryCount:1
A simple text message from the MQJMSIVT program
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
$>
$
```

Testing IBM MQ JMS client mode without JNDI

Carry out the following procedure:

1. Verify that the queue manager is started, and that the state of the queue manager is ACTIVE, by issuing the following command:

   ```
   WRKMQM MQMNAME(QMGRNAME)
   ```

2. Verify that the server connection channel is created, by issuing the following command:

   ```
   WRKMQMCHL CHLNAME( SYSTEM.DEF.SVRCONN ) CHLTYPE(*SVRCN)
   MQMNAME(QMGRNAME)
   ```

3. Verify that the listener is started for the correct port, by issuing the **WRKMQMLSR** command
4. Start the qshell, from the command line, by issuing the **STRQSH** command.
5. Verify that the CLASSPATH is correct by issuing the **export** command.
6. Use the **cd** command to change directory as follows:

   ```
   cd /qibm/proddata/mqm/java/bin
   ```

7. Run the script file:

```
IVTRun -nojndi -client -m QMgrName -host hostname [-port port] [-channel channel]
```

You receive output similar to the following example. Note that the copyright statements depend upon the versions of the products that you are using.

```
> IVTRun -nojndi -client -m ELCRTP19 -host ELCRTP19 -port 1414 -channel SYSTEM.DEF.SVRCONN

5724-H72, 5724-B41, 5655-F10 (c) Copyright IBM Corp. 2011, 2024.
All Rights Reserved.
WebSphere MQ classes for Java Message Service 5.300
Installation Verification Test


Creating a QueueConnectionFactory
Creating a Connection
Creating a Session
Creating a Queue
Creating a QueueSender
Creating a QueueReceiver
Creating a TextMessage
Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE
Reading the message back again

Got message:
JMS Message class: jms_text
JMSType: null
JMSDeliveryMode: 2
JMSExpiration: 0
JMSPriority: 4
JMSMessageID: ID:c1d4d840c5d3c3d9e3d7f1f9404040403ccf041f0000d012
JMSTimestamp: 1020274009970
JMSCorrelationID:null
JMSDestination: queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
JMSReplyTo: null
JMSRedelivered: false
JMS_IBM_PutDate:20040326
JMSXAppID:MQSeries Client for Java
JMS_IBM_Format:MQSTR
JMS_IBM_PutApplType:28
JMS_IBM_MsgType:8
JMSXUserID:QMQM
JMS_IBM_PutTime:14085237
JMSXDeliveryCount:1
A simple text message from the MQJMSIVT program
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
$
```

Testing IBM MQ JMS with JNDI

Verify that the queue manager is started, and that the state of the queue manager is ACTIVE, by issuing the following command:

```
WRKMQM MQMNAME(QMGRNAME)
```

Using the IVTRun sample test script

Carry out the following procedure:

1. Make the appropriate changes to the `JMSAdmin.config` file. To edit this file use the **EDTF** (Edit File) command from an IBM i command line

```
EDTF '/qibm/proddata/mqm/java/bin/JMSAdmin.config'
```

.

a. To use LDAP for Weblogic, remove the comment from:

```
INITIAL_CONTEXT_FACTORY=com.sun.jndi.ldap.LdapCtxFactory
```

b. To use LDAP for WebSphere Application Server, remove the comment from:

```
INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory
```

c. To test the file system, remove the comment from:

```
INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
```

d. Ensure that you have selected the correct PROVIDER_URL, by removing the comment from the appropriate line.

e. Comment out all other lines using the # symbol.

f. When you have completed all your changes, press **F2=Save** and **F3=Exit**.

2. Start the qshell, from the command line, by issuing the **STRQSH** command.
3. Verify that the CLASSPATH is correct by issuing the **export** command.
4. Use the **cd** command to change directory as follows:

```
cd /qibm/proddata/mqm/java/bin
```

5. Start the **IVTSetup** script to create the administered objects (*MQQueueConnectionFactory* and *MQQueue*), by issuing the **IVTSetup** command.
6. Run the IVTRun script by issuing the following command:

```
IVTRun -url providerURL [-icf initCtxFact]
```

You receive output similar to the following example. Note that the copyright statements depend upon the versions of the products that you are using.

```
> IVTSetup
+ Creating script for object creation within JMSAdmin
+ Calling JMSAdmin in batch mode to create objects
Ignoring unknown flag: -i

5724-H72 (c) Copyright IBM Corp. 2011, 2024. All Rights Reserved.
Starting WebSphere MQ classes for Java Message Service Administration

InitCtx>
InitCtx>
InitCtx>
InitCtx>
InitCtx>
Stopping MQSeries classes for Java Message Service Administration

+ Administration done; tidying up files
+ Done!
$
> IVTRun -url file:////tmp/mqjms -icf com.sun.jndi.fscontext.RefFSContextFactory

5724-H72 (c) Copyright IBM Corp. 2011, 2024. All Rights Reserved.
MQSeries classes for Java Message Service
Installation Verification Test

Using administered objects, please ensure that these are available

Retrieving a QueueConnectionFactory from JNDI
Creating a Connection
Creating a Session
Retrieving a Queue from JNDI
Creating a QueueSender
Creating a QueueReceiver
Creating a TextMessage
Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE
```

```
Reading the message back again

Got message:
JMS Message class: jms_text
JMSType: null
JMSDeliveryMode: 2
JMSExpiration: 0
JMSPriority: 4
JMSMessageID: ID:c1d4d840c5d3c3d9e3d7f1f9404040403ccf041f0000e012
JMSTimestamp: 1020274903770
JMSCorrelationID:null
JMSDestination: queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
JMSReplyTo: null
JMSRedelivered: false
JMS_IBM_Format:MQSTR
JMS_IBM_PutApplType:8
JMSXDeliveryCount:1
JMS_IBM_MsgType:8
JMSXUserID:STANLEY
JMSXAppID:QP0ZSPWT STANLEY 170308
A simple text message from the MQJMSIVT program
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
$
```

Java application development using a Maven repository

When developing a Java application for IBM MQ, by using a Maven repository to automatically install dependencies, you do not need to explicitly install anything before using IBM MQ interfaces.

Maven Central Repository

Maven is a tool for building applications and also provides a repository for holding artifacts that your application may want to access.

The Maven Repository (or Central Repository) has a structure that allows files such as JAR files to have distinct versions that are then easily discovered with a well-known naming mechanism. Build tools can then use those names to dynamically pull in the dependencies for your application. In the definition of your application, which, when using Maven as a build tool, is called the POM file, you name the dependencies and the build process knows what to do from there.

IBM MQ client files

Copies of the IBM MQ Java client interfaces are available in the Central Repository under the `com.ibm.mq GroupId`. You can find the `com.ibm.mq.jakarta.client.jar` file (Jakarta Messaging 3.0) and the `com.ibm.mq.allclient.jar` file (JMS 2.0). These files are typically used for standalone programs. You can also find the `wmq.jakarta.jmsra.rar` file (Jakarta Messaging 3.0) and the `wmq.jmsra.rar` file (JMS 2.0), which is for use in Java EE application servers). The `jakarta.client.jar` and the `allclient.jar` both contain the IBM MQ classes for JMS and the IBM MQ classes for Java.

Important: Using the Apache Maven Assembly Plugin *jar-with-dependencies* format to build an application which includes the IBM MQ relocatable JAR file is not supported.

In a `pom.xml` file processed by the maven command, you add dependencies for these JAR files as shown in the following examples:

- JMS 3.0 To show the relationship between your application code and `com.ibm.mq.jakarta.client.jar`:

```
<dependency>
        <groupId>com.ibm.mq</groupId>
        <artifactId>com.ibm.mq.jakarta.client</artifactId>
```

```
        <version>9.3.0.0</version>
    </dependency>
```

- JMS 2.0 To show the relationship between your application code and `com.ibm.mq.allclient.jar`:

```
<dependency>
        <groupId>com.ibm.mq</groupId>
        <artifactId>com.ibm.mq.allclient</artifactId>
        <version>9.2.2.0</version>
    </dependency>
```

- JMS 3.0 For using the Jakarta EE resource adapter:

```
<dependency>
     <groupId>com.ibm.mq</groupId>
     <artifactId>wmq.jakarta.jmsra</artifactId>
     <version>9.3.0.0</version>
 </dependency>
```

- JMS 2.0 For using the JMS 2.0 Java EE resource adapter:

```
<dependency>
     <groupId>com.ibm.mq</groupId>
     <artifactId>wmq.jmsra</artifactId>
     <version>9.2.2.0</version>
 </dependency>
```

For an example of a simple project in Eclipse to run a JMS project, see the IBM Developer article Developing Java applications for MQ just got easier with Maven.

Developing C++ applications

IBM MQ provides C++ classes equivalent to IBM MQ objects and some additional classes equivalent to the array data types. It provides a number of features not available through the MQI.

IBM WebSphere MQ 7.0, enhancements to the IBM MQ programming interfaces are not applied to the C++ classes.

IBM MQ C++ provides the following features:

- Automatic initialization of IBM MQ data structures.
- Just-in-time queue manager connection and queue opening.
- Implicit queue closure and queue manager disconnection.
- Dead-letter header transmission and receipt.
- IMS bridge header transmission and receipt.
- Reference message header transmission and receipt.
- Trigger message receipt.
- CICS bridge header transmission and receipt.
- Work header transmission and receipt.
- Client channel definition.

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

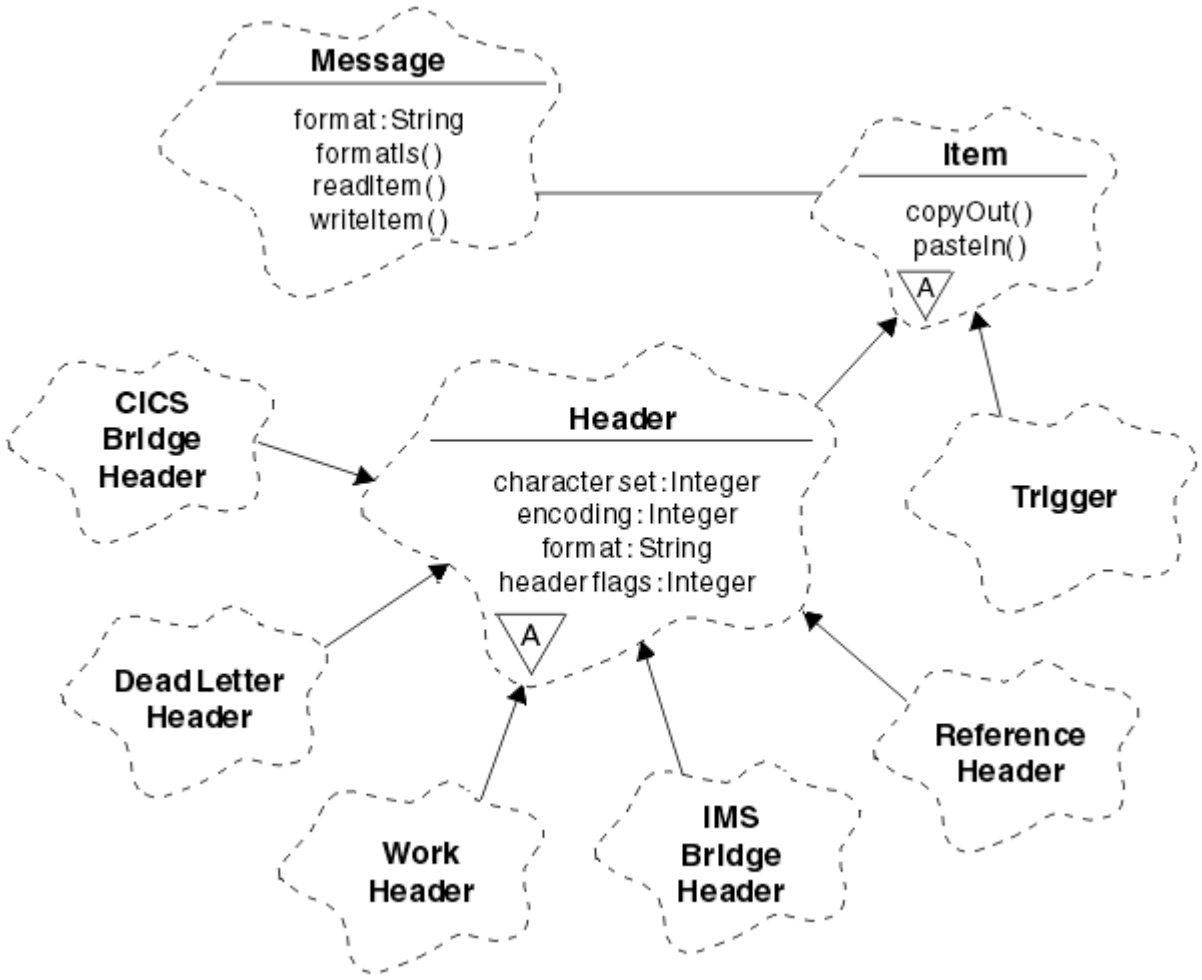


Figure 50. IBM MQ C++ classes (item handling)

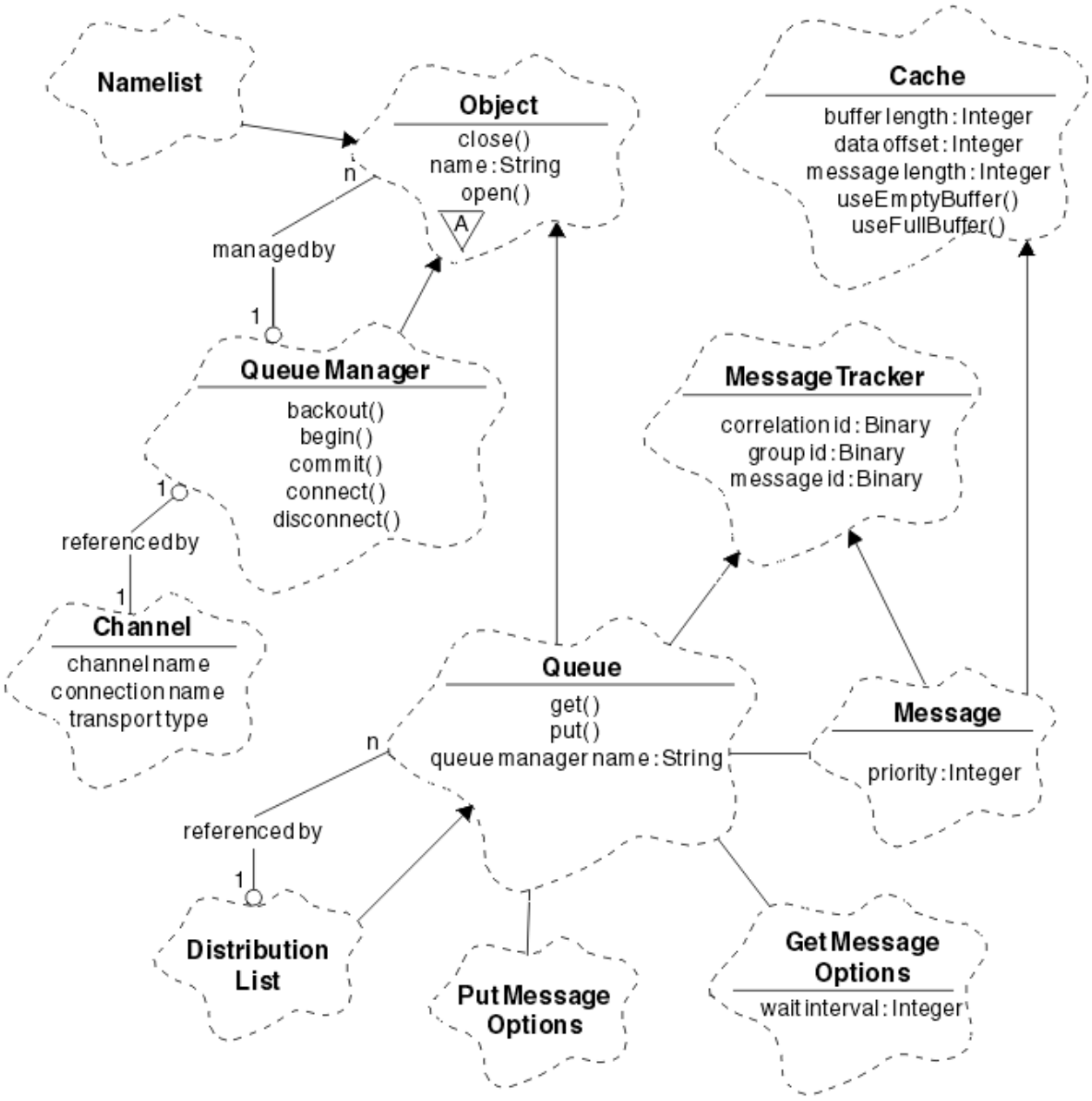


Figure 51. IBM MQ C++ classes (queue management)

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

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

The following classes and data types are used in the C++ method signatures of the queue management classes (see Figure 51 on page 510) and the item handling classes (see Figure 50 on page 509):

- The ImqBinary class (see ImqBinary C++ class), which encapsulates byte arrays such as MQBYTE24.
- The ImqBoolean data type, which is defined as **typedef unsigned char ImqBoolean**.

- The ImqString class (see ImqString C++ class), which encapsulates character arrays such as MQCHAR64.

Entities with data structures are subsumed within appropriate object classes. Individual data structure fields (see C++ and MQI cross-reference) are accessed with methods.

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

The ImqMessage class (see ImqMessage C++ class) encapsulates the MQMD data structure and also acts as a holding point for user data and *items* (see “Reading messages in C++” on page 520) by providing cached buffer facilities. You can provide fixed-length buffers for user data and use the buffer many times. The amount of data present in the buffer can vary from one use to the next. Alternatively, the system can provide and manage a buffer of flexible length. Both the size of the buffer (the amount available for receipt of messages) and the amount actually used (either the number of bytes for transmission or the number of bytes actually received) become important considerations.

Related concepts

Technical overview

“C++ sample programs” on page 511
Four sample programs are supplied, to demonstrate getting and putting messages.

“C++ language considerations” on page 515
This collection of topics details the aspects of the C++ language usage and conventions that you must consider when writing application programs that use the Message Queue Interface (MQI).

“Preparing message data in C++” on page 519
Message data is prepared in a buffer, which can be supplied by the system or the application. There are advantages to either method. Examples of using a buffer are given.

“Developing applications for IBM MQ” on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

Related reference

“Building IBM MQ C++ programs” on page 525
The URL of supported compilers is listed, together with the commands to use to compile, link and run C++ programs and samples on IBM MQ platforms.

C++ and MQI cross-reference

IBM MQ C++ classes

C++ sample programs

Four sample programs are supplied, to demonstrate getting and putting messages.

The sample programs are:

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

The sample programs are located in the directories shown in Table 73 on page 512.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Table 73. Location of sample programs		
Environment	**Directory containing source**	**Directory containing built programs**
AIX AIX	*MQ_INSTALLATION_PATH*/samp	*MQ_INSTALLATION_PATH*/samp/bin/ia
V 9.4.0 AIX AIX	*MQ_INSTALLATION_PATH*/samp	*MQ_INSTALLATION_PATH*/samp/bin/ca (see note "1" on page 512)
IBM i IBM i	`/QIBM/ProdData/mqm/samp/`	(see note "2" on page 512)
Linux Linux	*MQ_INSTALLATION_PATH*/samp	None
Windows Windows	*MQ_INSTALLATION_PATH*\tools\cplus\samples	*MQ_INSTALLATION_PATH*\tools\cplus\samples\bin\vn (see note "3" on page 512)
z/OS z/OS	thlqual.SCSQCPPS	

Notes:

1. V 9.4.0 AIX Programs built using the XLC 17 compiler are located in the "`ca`" folder, whilst programs built using the XLC 16 compiler are located in the "`ia`" folder.
2. IBM i Programs built using the ILE C++ compiler for IBM i are in the library QMQM. The source files are in `/QIBM/ProdData/mqm/samp`.
3. Windows Programs built using the Microsoft Visual Studio Visual Studio are found in *MQ_INSTALLATION_PATH*`\tools\cplus\samples\bin\vn`. For further information about these compilers, see "Building C++ programs on Windows" on page 531.

Sample program HELLO WORLD (imqwrld.cpp)

This C++ sample program shows how to put and get a regular datagram (C structure) using the ImqMessage class.

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

On all platforms except z/OS

If you are using a server connection to IBM MQ, follow one of the following procedures:

- To use the existing default queue, SYSTEM.DEFAULT.LOCAL.QUEUE, run the program **`imqwrlds`** without passing any parameters
- To use a temporary dynamically assigned queue, run **`imqwrlds`** passing the name of the default model queue, SYSTEM.DEFAULT.MODEL.QUEUE.

If you are using a client connection to IBM MQ, follow one of the following procedures:

- Set up the MQSERVER environment variable (see *MQSERVER* for more information) and run **`imqwrldc`**, or

- Run **imqwrldc** passing as parameters the **queue-name**, **queue-manager-name**, and **channel-definition**, where a typical **channel-definition** might be SYSTEM.DEF.SVRCONN/TCP/*hostname* (1414)

On z/OS

z/OS

Construct and run a batch job, using the sample JCL **imqwrldr**.

See z/OS Batch, RRS Batch and CICS for more information.

Sample code

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

#include <imqi.hpp> // IBM MQ C++

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

#define BUFFER_SIZE 12

static char gpszHello[ BUFFER_SIZE ] = "Hello world" ;
int main ( int argc, char * * argv ) {
  ImqQueueManager manager ;
  int iReturnCode = 0 ;

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

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


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

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

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

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

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

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

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

      if ( pqueue -> get( * pmsg ) ) {
        int iDataLength = pmsg -> dataLength( );

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

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

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

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

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

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

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

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

  return iReturnCode ;
}
```

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

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

These samples show the use of the following classes:

- ImqError (see ImqError C++ class)
- ImqMessage (see ImqMessage C++ class)
- ImqObject (see ImqObject C++ class)
- ImqQueue (see ImqQueue C++ class)

- ImqQueueManager (see ImqQueueManager C++ class)

Follow the appropriate instructions to run the programs.

On all platforms except z/OS

1. Run **imqsputs** *queue-name*.
2. Type lines of text at the console. These lines are placed as messages onto the specified queue.
3. Enter a null line to end the input.
4. Run **imqsgets** *queue-name* to retrieve all the lines and display them at the console.

z/OS See “Building C++ programs on z/OS Batch, RRS Batch and CICS” on page 533 for more information.

On z/OS

z/OS

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

Sample program DPUT (imqdput.cpp)

This C++ sample program puts messages to a distribution list consisting of two queues.

DPUT shows the use of the ImqDistributionList class (see ImqDistributionList C++ class). This sample is not supported on z/OS.

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

C++ language considerations

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

C++ Header files

Header files are provided as part of the definition of the MQI, to help you write IBM MQ application programs in the C++ language.

These header files are summarized in the following table.

Table 74. C/C++ header files

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

To improve the portability of applications, code the name of the header file in lowercase on the **#include** preprocessor directive:

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

C++ methods and attributes

Method names are in mixed case. Various considerations apply to parameters and return values. Attributes are accessed using set and get methods as appropriate.

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

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

All method names in these classes are mixed case, beginning with lowercase. Each word, except the first within a method name, begins with a capital letter. Abbreviations are not used unless their meaning is widely understood. Abbreviations used include *id* (for identity) and *sync* (for synchronization).

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

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

Data types in C++

All data types are defined by the C **typedef** statement.

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

Manipulating binary strings in C++

Strings of binary data are declared as objects of the **ImqBinary** class. Objects of this class can be copied, compared, and set using the familiar C operators. Example code is provided.

The following code sample shows operations on a binary string:

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

ImqMessage message ;
ImqBinary id, correlationId ;
MQBYTE24 byteId ;

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

Manipulating character strings in C++

Character data is often returned in **ImqString** class objects which can be cast to **char *** using a conversion operator. The ImqString class contains methods to assist in the processing of character strings.

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

Note: The **char *** conversion result from an **ImqString** class object might be null.

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

Initial state of objects in C++

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

Using C from C++

When you use C functions from a C++ program, include appropriate headers.

The following example shows `string.h` included in a C++ program:

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

C++ notational conventions

This example shows how to invoke methods and declare parameters.

This code sample uses the methods and parameters **ImqBoolean ImqQueue**:: **get** (**ImqMessage &** *msg*)

Declare and use the parameters as follows:

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

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

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

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

...
}
```

Implicit operations in C++

Several operations can occur implicitly, *just in time*, to satisfy the prerequisite conditions for the successful execution of a method. These implicit operations are connect, open, reopen, close, and disconnect. You can control connect and open implicit behavior using class attributes.

Connect

An ImqQueueManager object is connected automatically for any method that results in any call to the MQI (see C++ and MQI cross-reference).

Open

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

Reopen

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

Reopen can cause problems in specific circumstances:

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

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

Setting the **open options** explicitly for complex queue-handling situations results in better performance and avoids the problems associated with the use of reopen.

Close

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

Disconnect

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

Binary and character strings in C++

The ImqString class encapsulates the traditional *char ** data format. The ImqBinary class encapsulates the binary byte array. Some methods that set character data might truncate the data.

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

The ImqString class (see ImqString C++ class) encapsulates the traditional **char *** and provides support for:

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

The ImqBinary class (see ImqBinary C++ class) encapsulates binary byte arrays of arbitrary size. In particular, it is used to hold the following attributes:

- **accounting token** (MQBYTE32)
- **connection tag** (MQBYTE128)
- **correlation id** (MQBYTE24)
- **facility token** (MQBYTE8)
- **group id** (MQBYTE24)

- **instance id** (MQBYTE24)
- **message id** (MQBYTE24)
- **message token** (MQBYTE16)
- **transaction instance id** (MQBYTE16)

Where these attributes belong to objects of the following classes:

- ImqCICSBridgeHeader (see ImqCICSBridgeHeader C++ class)
- ImqGetMessageOptions (see ImqGetMessageOptions C++ class)
- ImqIMSBridgeHeader (see ImqIMSBridgeHeader C++ class)
- ImqMessageTracker (see ImqMessageTracker C++ class)
- ImqQueueManager (see ImqQueueManager C++ class)
- ImqReferenceHeader (see ImqReferenceHeader C++ class)
- ImqWorkHeader (see ImqWorkHeader C++ class)

The ImqBinary class also provides support for comparison and copying.

Unsupported functions in C++

The IBM MQ C++ classes and methods are independent of IBM MQ platform. They might therefore offer some functions that are not supported on certain platforms.

If you try to use a function on a platform on which it is not supported, the function is detected by IBM MQ but not by the C++ language bindings. IBM MQ reports the error to your program, like any other MQI error.

Messaging in C++

This collection of topics details how to prepare, read, and write messages in C++.

Preparing message data in C++

Message data is prepared in a buffer, which can be supplied by the system or the application. There are advantages to either method. Examples of using a buffer are given.

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

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

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

The following examples show simplified message sends.

1. Use prepared data in a user-supplied buffer

```
char szBuffer[ ] = "Hello world" ;

msg.useFullBuffer( szBuffer, sizeof( szBuffer ) );
msg.setFormat( MQFMT_STRING );
```

2. Use prepared data in a user-supplied buffer, where the buffer size exceeds the data size

```
char szBuffer[ 24 ] = "Hello world" ;

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

3. Copy data to a user-supplied buffer

```
char szBuffer[ 12 ];

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

4. Copy data to a system-supplied buffer

```
msg.setFormat( MQFMT_STRING );
msg.write( 12, "Hello world" );
```

5. Copy data to a system-supplied buffer using objects (objects set the message format as well as content)

```
ImqString strText( "Hello world" );

msg.writeItem( strText );
```

Reading messages in C++

A buffer can be supplied by the application or the system. Data can be accessed directly from the buffer or read sequentially. There is a class equivalent to each message type. Sample code is given.

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

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

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

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

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

```
ImqQueue queue ;
ImqMessage msg ;

if ( queue.get( msg ) ) {

  /* Process all items of data in the message buffer. */
```

```
  do while ( msg.dataLength( ) ) {
    ImqBoolean bFormatKnown = FALSE ;
    /* There remains unprocessed data in the message buffer. */

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

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

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

        /* Process the information in the dead-letter object.  */
        /* Note that the encoding and character set have       */
        /* already been processed.                             */
        ...
      }
      /* There might be another item after this, */
      /* or just the user data.                  */
    }
    if ( msg.formatIs( MQFMT_TRIGGER ) ) {
      ImqTrigger trigger ;
      /* The next item is a trigger message.                   */
      /* For the next statement to work and return TRUE,       */
      /* the correct class of object pointer must be supplied. */
      bFormatKnown = TRUE ;
      if ( msg.readItem( trigger ) ) {

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

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

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

      if ( msg.readItem( object ) ) {
        /* The user-defined data has been extricated from the */
        /* buffer and transformed into a user-defined object. */

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

      /* Continue looking for further items. */
    }
    if ( ! bFormatKnown ) {
      /* There remains data that is not associated with a specific*/
      /* item class.                                               */
      char * pszDataPointer = msg.dataPointer( );       /* Address.*/
      int iDataLength = msg.dataLength( );              /* Length. */

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

    }

  }
}
```

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

UserClass is derived from the ImqItem class (see ImqItem C++ class), and implements the virtual **copyOut** and **pasteIn** methods from that class.

The next two examples show code from the ImqDeadLetterHeader class (see ImqDeadLetterHeader C++ class). The first example shows custom-encapsulated message- *writing* code.

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

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

  return bSuccess ;
}
```

The second example shows custom-encapsulated message- *reading* code.

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

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

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

          // Reflect the cache error in this object.
          setReasonCode( msg.reasonCode( ) );
          setCompletionCode( msg.completionCode( ) );
        }
      } else {
```

```
        setReasonCode( MQRC_INCONSISTENT_FORMAT );
        setCompletionCode( MQCC_FAILED );
      }
    } else {
      setReasonCode( MQRC_ENCODING_ERROR );
      setCompletionCode( MQCC_FAILED );
    {
  } else {
    setReasonCode( MQRC_STRUC_ID_ERROR );
    setCompletionCode( MQCC_FAILED );
  }

  return bSuccess ;
}
```

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

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

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

The next example shows how a fixed area of storage can be used to receive messages:

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

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

delete [ ] pszBuffer ;
```

In this code fragment, the buffer can always be addressed directly, with *pszBuffer*, as opposed to using the **bufferPointer** method. However, it is better to use the **dataPointer** method for general-purpose access. The application (not the ImqCache class object) must discard a user-defined (nonautomatic) buffer.

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

Writing a message to the dead-letter queue in C++

Example program code for writing a message to the dead-letter queue.

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

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

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

// Set up the dead-letter header information.
header.setDestinationQueueManagerName( mgr.name( ) );
header.setDestinationQueueName( queueIn.name( ) );
header.setPutApplicationName( /* ? */ );
header.setPutApplicationType( /* ? */ );
```

```
header.setPutDate( /* TODAY */ );
header.setPutTime( /* NOW */ );
header.setDeadLetterReasonCode( FB_APPL_ERROR_1234 );

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

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

Writing a message to the IMS bridge in C++

Example program code for writing a message to the IMS bridge.

Messages sent to the IBM MQ - IMS bridge might use a special header. The IMS bridge header is prefixed to regular message data.

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

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

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

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

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

Writing a message to the CICS bridge in C++

Example program code for writing a message to the CICS bridge.

Messages sent to IBM MQ for z/OS using the CICS bridge require a special header. The CICS bridge header is prefixed to regular message data.

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

```
ImqCicsBridgeHeader header ;  // CICS bridge header information.

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

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

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

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

Writing a message with a work header in C++

Example program code for writing a message destined for a queue managed by the z/OS Workload Manager.

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

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

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

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

// Send the message to the WLM managed queue.
queueWLM.setConnectionReference( mgr );
queueWLM.setName( /* ? */ );
queueWLM.put( msg );
```

Building IBM MQ C++ programs

The URL of supported compilers is listed, together with the commands to use to compile, link and run C++ programs and samples on IBM MQ platforms.

For a list of the compilers for each supported platform and version of IBM MQ, see System Requirements for IBM MQ.

The command you need to compile and link your IBM MQ C++ program depends on your installation and requirements. The examples that follow show typical compiling and linking commands for some of the compilers using the default installation of IBM MQ on a number of platforms.

AIX Building C++ programs on AIX

Build IBM MQ C++ programs on AIX using the XL C Enterprise Edition compiler.

V 9.4.0 For more information about the different mapping of compiler options between XLC 16 and XLC 17 compilers, see Mapping of options.

Deprecated V 9.4.0 Support for the XL C/C++ for AIX 16 compiler on AIX is deprecated from IBM MQ 9.4.0.

Client

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

32-bit unthreaded application

```
xlC -o imqsputc_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqc23ia -limqb23ia -lmqic
```

32-bit threaded application

```
xlC_r -o imqsputc_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqc23ia_r -limqb23ia_r -lmqic_r
```

64-bit unthreaded application

```
xlC -q64 -o imqsputc_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqc23ia -limqb23ia -lmqic
```

64-bit threaded application

```
xlC_r -q64 -o imqsputc_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqc23ia_r -limqb23ia_r -lmqic_r
```

V 9.4.0 32-bit unthreaded application (XLC 17)

```
ibm-clang++_r -o imqsputc_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqc23ca -limqb23ca -lmqic
```

V 9.4.0 32-bit threaded application (XLC 17)

```
ibm-clang++_r -o imqsputc_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqc23ca_r -limqb23ca_r -lmqic_r
```

V 9.4.0 64-bit unthreaded application (XLC 17)

```
ibm-clang++_r -m64 -o imqsputc_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqc23ca -limqb23ca -lmqic
```

V 9.4.0 64-bit threaded application (XLC 17)

```
ibm-clang++_r -m64 -o imqsputc_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqc23ca_r -limqb23ca_r -lmqic_r
```

Server

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

32-bit unthreaded application

```
xlC -o imqsput_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqs23ia -limqb23ia -lmqm
```

32-bit threaded application

```
xlC_r -o imqsput_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqs23ia_r -limqb23ia_r -lmqm_r
```

64-bit unthreaded application

```
xlC -q64 -o imqsput_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqs23ia -limqb23ia -lmqm
```

64-bit threaded application

```
xlC_r -q64 -o imqsput_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqs23ia_r -limqb23ia_r -lmqm_r
```

V 9.4.0 **32-bit unthreaded application (XLC 17)**

```
ibm-clang++_r -o imqsput_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqs23ca -limqb23ca -lmqm
```

V 9.4.0 **32-bit threaded application (XLC 17)**

```
ibm-clang++_r -o imqsput_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -limqs23ca_r -limqb23ca_r -lmqm_r
```

V 9.4.0 **64-bit unthreaded application (XLC 17)**

```
ibm-clang++_r -m64 -o imqsput_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqs23ca -limqb23ca -lmqm
```

V 9.4.0 **64-bit threaded application (XLC 17)**

```
ibm-clang++_r -m64 -o imqsput_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -limqs23ca_r -limqb23ca_r -lmqm_r
```

IBM i Building C++ programs on IBM i

Build IBM MQ C++ programs on IBM i using the ILE C++ compiler.

IBM ILE C++ for IBM i is a native compiler for C++ programs. The following instructions describe how to use this compiler to create IBM MQ C++ applications using the *Hello World!* IBM MQ sample program as an example.

1. Install the ILE C++ for IBM i compiler as directed in the *Read Me first!* manual that accompanies the product.
2. Ensure that the QCXXN library is in your library list.
3. Create the HELLO WORLD sample program:
   a. Create a module:

```
CRTCPPMOD MODULE(MYLIB/IMQWRLD) +
SRCSTMF('/QIBM/ProdData/mqm/samp/imqwrld.cpp') +
INCDIR('/QIBM/ProdData/mqm/inc') DFTCHAR(*SIGNED) +
TERASPACE(*YES)
```

   The source for the C++ sample programs can be found in `/QIBM/ProdData/mqm/samp` and the include files in `/QIBM/ProdData/mqm/inc`.

   Alternatively, the source can be found in library `SRCFILE(QCPPSRC/LIB) SRCMBR(IMQWRLD)`.

b. Bind this with IBM MQ-supplied service programs to produce a program object:

```
CRTPGM PGM(MYLIB/IMQWRLD) MODULE(MYLIB/IMQWRLD) +
BNDSRVPGM(QMQM/IMQB23I4 QMQM/IMQS23I4)
```

To build a threaded application use the re-entrant service programs:

```
CRTPGM PGM(MYLIB/IMQWRLD) MODULE(MYLIB/IMQWRLD) +
BNDSRVPGM(QMQM/IMQB23I4[_R] QMQM/IMQS23I4[_R])
```

c. Execute the HELLO WORLD sample program, using SYSTEM.DEFAULT.LOCAL.QUEUE:

```
CALL PGM(MYLIB/IMQWRLD)
```

Linux Building C++ programs on Linux

Build IBM MQ C++ programs on Linux using the GNU g++ compiler.

System p

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Client: System p

32-bit unthreaded application

```
g++ -m32 -o imqsputc_32 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqc23gl
-limqb23gl -lmqic
```

32-bit threaded application

```
g++ -m32 -o imqsputc_r32 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqc23gl_r
-limqb23gl_r -lmqic_r
```

64-bit unthreaded application

```
g++ -m64 -o imqsputc_64 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqc23gl -limqb23gl -lmqic
```

64-bit threaded application

```
g++ -m64 -o imqsputc_r64 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqc23gl_r -limqb23gl_r -lmqic_r
```

Server: System p

32-bit unthreaded application

```
g++ -m32 -o imqsput_32 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqs23gl
-limqb23gl -lmqm
```

32-bit threaded application

```
g++ -m32 -o imqsput_r32 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqs23gl_r
-limqb23gl_r -lmqm_r
```

64-bit unthreaded application

```
g++ -m64 -o imqsput_64 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqs23gl -limqb23gl -lmqm
```

64-bit threaded application

```
g++ -m64 -o imqsput_r64 imqsput.cpp -fsigned-char -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqs23gl_r -limqb23gl_r -lmqm_r
```

IBM Z

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Client: IBM Z

32-bit unthreaded application

```
g++ -m31 -fsigned-char -o imqsputc_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqc23gl -limqb23gl -lmqic
```

32-bit threaded application

```
g++ -m31 -fsigned-char -o imqsputc_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqc23gl_r -limqb23gl_r -lmqic_r
-lpthread
```

64-bit unthreaded application

```
g++ -m64 -fsigned-char -o imqsputc_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqc23gl -limqb23gl -lmqic
```

64-bit threaded application

```
g++ -m64 -fsigned-char -o imqsputc_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqc23gl_r -limqb23gl_r -lmqic_r -lpthread
```

Server: IBM Z

32-bit unthreaded application

```
g++ -m31 -fsigned-char -o imqsput_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqs23gl -limqb23gl -lmqm
```

32-bit threaded application

```
g++ -m31 -fsigned-char -o imqsput_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib
-limqs23gl_r -limqb23gl_r -lmqm_r -lpthread
```

64-bit unthreaded application

```
g++ -m64 -fsigned-char -o imqsput_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqs23gl -limqb23gl -lmqm
```

64-bit threaded application

```
g++ -m64 -fsigned-char -o imqsput_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-limqs23gl_r -limqb23gl_r -lmqm_r -lpthread
```

x86-64 (32-bit)

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Client: x86-64 (32-bit)

32-bit unthreaded application

```
g++ -m32 -fsigned-char -o imqsputc_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -L
MQ_INSTALLATION_PATH/lib -Wl,
-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib -limqc23gl -limqb23gl -lmqic
```

32-bit threaded application

```
g++ -m32 -fsigned-char -o imqsputc_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -L MQ_INSTALLATION_PATH/lib
-Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib -limqc23gl_r -limqb23gl_r
-lmqic_r -lpthread
```

64-bit unthreaded application

```
g++ -m64 -fsigned-char -o imqsputc_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -L
MQ_INSTALLATION_PATH/lib64
-Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64 -limqc23gl -limqb23gl
-lmqic
```

64-bit threaded application

```
g++ -m64 -fsigned-char -o imqsputc_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -L
MQ_INSTALLATION_PATH/lib64
-Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64 -limqc23gl_r -limqb23gl_r
-lmqic_r -lpthread
```

Server: x86-64 (32-bit)

32-bit unthreaded application

```
g++ -m32 -fsigned-char -o imqsput_32 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -L MQ_INSTALLATION_PATH/lib
-Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib -limqs23gl -limqb23gl -lmqm
```

32-bit threaded application

```
g++ -m32 -fsigned-char -o imqsput_32_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH lib -Wl,-rpath= MQ_INSTALLATION_PATH/lib -L MQ_INSTALLATION_PATH/lib
-Wl,-rpath= MQ_INSTALLATION_PATH/lib -Wl,-rpath=/usr/lib -limqs23gl_r -limqb23gl_r
-lmqm_r -lpthread
```

64-bit unthreaded application

```
g++ -m64 -fsigned-char -o imqsput_64 imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -L
MQ_INSTALLATION_PATH/lib64
-Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64 -limqs23gl -limqb23gl -lmqm
```

64-bit threaded application

```
g++ -m64 -fsigned-char -o imqsput_64_r imqsput.cpp -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -L
MQ_INSTALLATION_PATH/lib64
-Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64 -limqs23gl_r -limqb23gl_r
-lmqm_r -lpthread
```

Windows Building C++ programs on Windows

Build IBM MQ C++ programs on Windows by using the Microsoft Visual Studio C++ compiler.

Attention: The libraries shipped by IBM MQ are dynamic libraries and not static libraries. IBM MQ provides something known as "`import libraries`" that you can use during compilation time only. For runtime, you must use the dynamic libraries.

From IBM MQ 8.0.0 Fix Pack 4, IBM MQ ships redistributable clients, containing libraries required for running IBM MQ applications. These libraries can be packaged and redistributed with client applications. For more information, see Redistributable clients on Windows.

Library (.lib) files and dll files for use with 32-bit applications are installed in *MQ_INSTALLATION_PATH*`/Tools/Lib`. Files for use with 64-bit applications are installed in *MQ_INSTALLATION_PATH*`/Tools/Lib64`. *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

Client

```
cl -MD imqsput.cpp /Feimqsputc.exe imqb23vn.lib imqc23vn.lib
```

Server

```
cl -MD imqsput.cpp /Feimqsput.exe imqb23vn.lib imqs23vn.lib
```

Installing the universal C runtime

If you are using Windows 8.1 or Windows Server 2012 R2, you must install the universal C runtime update (Universal CRT) from Microsoft. This runtime is included as part of Windows 10, and Windows Server 2016.

The Universal CRT update is Microsoft update KB3118401. You can check to see if you have this update by searching for a file that is called `ucrtbase.dll` in your `C:\Windows\System32` directory. If not, you can download the update from the following Microsoft page: https://www.catalog.update.microsoft.com/Search.aspx?q=kb3118401.

Attempting to run an IBM MQ program, or a program you compile yourself using Microsoft Visual Studio 2017, without the runtime installed results in errors such as the following error:

```
The program can't start because api-ms-win-crt-runtime-|1-1-0.dll
is missing from your computer. Try reinstalling the program to
fix this problem.
```

Providing runtimes for Microsoft Visual Studio 2012 programs

If you compiled an IBM MQ program by using Microsoft Visual Studio 2012, be aware that the IBM MQ installer does not install the Microsoft Visual Studio 2012 C/C++ runtimes. If your previous version of IBM MQ was installed on the same computer, the Microsoft Visual Studio 2012 runtimes are available from that installation.

However, if you are using a program that was built by using Microsoft Visual Studio 2012 and no previous version of IBM MQ was installed, you must do one of the following things:

- Download and install the **Microsoft Visual C++ Redistributable for VisualStudio 2017 (32 and 64-bit versions)** from Microsoft.
- Recompile your program with Microsoft Visual Studio 2017, or another Microsoft Visual Studio level for which the runtimes are installed.

C++ client libraries built by using the Microsoft Visual Studio 2015 compiler

IBM MQ provides C++ client libraries that are built with the Microsoft Visual Studio 2015 C++ compiler, and the Microsoft Visual Studio 2017 C++ compiler.

Both 32-bit and 64-bit versions of the IBM MQ C++ libraries are provided. The 32-bit libraries are installed under the `bin\vs2015` folder, and the 64-bit libraries are installed under the `bin64\vs2015` folders.

By default, IBM MQ is configured to use the Microsoft Visual Studio 2017 libraries. To use the Microsoft Visual Studio 2015 libraries you must set the `MQ_PREFIX_VS_LIBRARIES` environment variable to `MQ_PREFIX_VS_LIBRARIES=vs2015` before you install IBM MQ, or before you use the **setmqenv** or **setmqinst** command.

Using differently named IBM MQ C++ libraries

IBM MQ provides some additional C++ client libraries that are named differently. These libraries are built with the Microsoft Visual Studio 2015 and Microsoft Visual Studio 2017 C++ compilers. These libraries are provided in addition to the existing C++ libraries that are also built with the Microsoft Visual Studio 2017 C++ compiler. Since these additional IBM MQ C++ libraries have different names, you can run IBM MQ C++ applications that are built by using IBM MQ C++ and compiled with Microsoft Visual Studio 2017 and earlier versions of the product on the same computer.

The additional Microsoft Visual Studio 2017 libraries have the following names:

- `imqb23vnvs2017.dll`
- `imqc23vnvs2017.dll`
- `imqs23vnvs2017.dll`
- `imqx23vnvs2017.dll`

The additional Microsoft Visual Studio 2015 libraries have the following names:

- `imqb23vnvs2015.dll`
- `imqc23vnvs2015.dll`
- `imqs23vnvs2015.dll`
- `imqx23vnvs2015.dll`

Both 32-bit and 64-bit versions of these libraries are provided. The 32-bit libraries are installed under the `bin` folder, and the 64-bit libraries are installed under the `bin64` folder. Corresponding import libraries are installed under the `Tools\lib` and `Tools\lib64` directories.

If your application uses `imq*vs2015.lib` files, you must compile it using the Microsoft Visual Studio 2015 compiler. To run IBM MQ C++ applications that are compiled with Microsoft Visual Studio 2015, or applications that are compiled with an earlier version of the product on the same computer, the PATH environment variable must be prefixed as shown in the following examples:

- For 32-bit applications:

```
SET PATH=installation folder\bin\vs2015;%PATH%
```

- For 64-bit applications:

```
SET PATH=installation folder\bin64\vs2015;%PATH%
```

Related concepts

Windows: changes from IBM MQ 8.0

z/OS Building C++ programs on z/OS Batch, RRS Batch and CICS

Build IBM MQ C++ programs on z/OS for the Batch, RRS batch or CICS environments and run the sample programs.

You can write C++ programs for three of the environments that IBM MQ for z/OS supports:

- Batch
- RRS batch
- CICS

Compile, prelink and link

Create an z/OS application by compiling, pre-linking, and link-editing your C++ source code.

IBM MQ C++ for z/OS is implemented as z/OS DLLs for the IBM C++ for z/OS language. Using DLLs, you concatenate the supplied definition sidedecks with the compiler output at pre-link time. This allows the linker to check your calls to the IBM MQ C++ member functions.

Note: There are three sets of sidedecks for each of the three environments.

To build an IBM MQ for z/OS C++ application, create and run JCL. Use the following procedure:

1. If your application runs under CICS, use the CICS-supplied procedure to translate CICS commands in your program.

   In addition, for CICS applications you need to:

   a. Add the SCSQLOAD library to the DFHRPL concatenation.

   b. Define the CSQCAT1 CEDA group using the member IMQ4B100 in the SCSQPROC library.

   c. Install CSQCAT1.

2. Compile the program to produce object code. The JCL for your compilation must include statements that make the product data definition files available to the compiler. The data definitions are supplied in the following IBM MQ for z/OS libraries:

   - **thlqual**.SCSQC370
   - **thlqual**.SCSQHPPS

   Be sure to specify the `/cxx` compiler option.

   Note: The name **thlqual** is the high level qualifier of the IBM MQ installation library on z/OS.

3. Pre-link the object code created in step “2” on page 533, including the following definition sidedecks, which are supplied in **thlqual**.SCSQDEFS:
   a. imqs23dm and imqb23dm for batch
   b. imqs23dr and imqb23dr for RRS batch
   c. imqs23dc and imqb23dc for CICS

   These are the corresponding DLLs.
   a. imqs23im and imqb23im for batch
   b. imqs23ir and imqb23ir for RRS batch
   c. imqs23ic and imqb23ic for CICS
4. Link-edit the object code created in step “3” on page 534, to produce a load module, and store it in your application load library.

To run batch or RRS batch programs, include the libraries **thlqual**.SCSQAUTH and **thlqual**.SCSQLOAD in the STEPLIB or JOBLIB data set concatenation.

To run a CICS program, first get your system administrator to define it to CICS as an IBM MQ program and transaction. You can then run it in the usual way.

Run the sample programs

The programs are described in “C++ sample programs” on page 511.

The sample applications are supplied in source form only. The files are:

Table 75. z/OS sample program files

Sample	Source program (in library thlqual.SCSQCPPS)	JCL (in library thlqual.SCSQPROC)
HELLO WORLD	imqwrld	imqwrldr
SPUT	imqsput	imqsputr
SGET	imqsget	imqsgetr

To run the samples, compile and link-edit them as with any C++ program (see “Building C++ programs on z/OS Batch, RRS Batch and CICS” on page 533). Use the supplied JCL to construct and run a batch job. You must initially customize the JCL, by following the commentary included with it.

z/OS Building C++ programs on z/OS UNIX System Services

Build IBM MQ C++ programs on z/OS UNIX System Services (z/OS UNIX).

To build an application under the z/OS UNIX shell, you must give the compiler access to the IBM MQ include files (located in `thlqual.SCSQC370` and `hlqual.SCSQHPPS`), and link against two of the DLL sidedecks (located in `thlqual.SCSQDEFS`). At runtime, the application needs access to the IBM MQ data sets `thlqual.SCSQLOAD`, `thlqual.SCSQAUTH`, and one of the language specific data sets, such as `thlqual.SCSQANLE` [6].

Compiling

1. Copy the sample into the file system using the TSO **oput** command, or use FTP. The rest of this example assumes that you have copied the sample into a directory called `/u/fred/sample`, and named it `imqwrld.cpp`.
2. Log into the z/OS UNIX shell, and change to the directory where you placed the sample.

[6] You can link with any of the sidedecks listed in "Pre-link the object code to run your z/OS UNIX in any of the three environments, “Building C++ programs on z/OS Batch, RRS Batch and CICS” on page 533

3. Set up the C++ compiler so that it can accept the DLL sidedeck and .cpp files as input:

```
/u/fred/sample:> export _CXX_EXTRA_ARGS=1
/u/fred/sample:> export _CXX_CXXSUFFIX="cpp"
```

4. Compile and link the sample program. The following command links the program with the batch sidedecks; the RRS batch sidedecks can be used instead. The \ character is used to split the command over more than one line. Do not enter this character; enter the command as a single line:

```
/u/fred/sample:> c++ -o imqwrld -I "//'thlqual.SCSQC370'" \
-I "//'thlqual.SCSQHPPS'" imqwrld.cpp \
"//'thlqual.SCSQDEFS(IMQS23DM)'" "//'thlqual.SCSQDEFS(IMQB23DM)'"
```

For more information on the TSO **oput** command, refer to the z/OS UNIX Command Reference.

You can also use the make utility to simplify building C++ programs. Here is a sample makefile to build the HELLO WORLD C++ sample program. It separates the compiling and linking stages. Set up the environment as in step "3" on page 535 before running make.

```
flags = -I "//'thlqual.SCSQC370'" -I "//'thlqual.SCSQHPPS'"
decks = "//'thlqual.SCSQDEFS(IMQS23DM)'" "//'thlqual.SCSQDEFS(IMQB23DM)'"

imqwrld: imqwrld.o
   c++ -o imqwrld imqwrld.o $(decks)

imqwrld.o: imqwrld.cpp
   c++ -c -o imqwrld $(flags) imqwrld.cpp
```

Refer to z/OS UNIX System Services Programming Tools for more information on using make.

Running

1. Log into the z/OS UNIX shell, and change to the directory where you built the sample.
2. Set up the STEPLIB environment variable to include the IBM MQ data sets:

```
/u/fred/sample:> export STEPLIB=$STEPLIB:thlqual.SCSQLOAD
/u/fred/sample:> export STEPLIB=$STEPLIB:thlqual.SCSQAUTH
/u/fred/sample:> export STEPLIB=$STEPLIB:thlqual.SCSQANLE
```

3. Run the sample:

```
/u/fred/sample:> ./imqwrld
```

Developing .NET applications

IBM MQ classes for .NET allow .NET applications to connect to IBM MQ as an IBM MQI client or to connect directly to an IBM MQ server.

Before you begin

Deprecated V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, in IBM MQ classes for .NET, the methods WriteObject(), ReadObject(), CreateObjectMessage(), and the classes ObjectMessage and XmsObjectMessageImpl used for serialization and deserialization of data are deprecated.

Removed V 9.4.0 V 9.4.0 The IBM MQ .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

About this task

IBM MQ classes for .NET is a set of classes that enable .NET applications to interact with IBM MQ. They represent the various components of IBM MQ which your application uses, such as queue managers, queues, channels and messages. For more information about these classes, see The IBM MQ .NET classes and interfaces.

V 9.4.0 IBM MQ 9.4.0 provides an IBM MQ .NET client library built against .NET 6 as the target framework. For more information, see “Installing IBM MQ classes for .NET” on page 537.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for .NET. For more information, see “Installing IBM MQ classes for .NET” on page 537.

If you have applications that use Microsoft .NET Framework and want to take advantage of the facilities of IBM MQ, you must use IBM MQ classes for .NET Framework. For more information, see “Installing IBM MQ classes for .NET Framework” on page 543.

For more information about the differences between IBM MQ classes for .NET Framework and IBM MQ classes for .NET, see “Installing IBM MQ classes for .NET” on page 537.

IBM MQ .NET managed applications are able to automatically balance connections across clustered queue managers. Both the IBM MQ classes for .NET and the IBM MQ classes for .NET Framework libraries are supported. For more information, see About uniform clusters and Automatic application balancing.

The object-oriented IBM MQ .NET interface is different from the MQI interface in that it uses methods of objects rather than using the MQI verbs. The procedural IBM MQ application programming interface is built around verbs such as those in the following list:

```
MQCONN, MQDISC, MQOPEN, MQCLOSE,
MQINQ, MQSET, MQGET, MQPUT, MQSUB
```

These verbs all take, as a parameter, a handle to the IBM MQ object on which they are to operate. Because .NET is object-oriented, the .NET programming interface turns this round. Your program consists of a set of IBM MQ objects, which you act upon by calling methods on those objects. You can write programs in any language supported by .NET.

When you use the procedural interface, you disconnect from a queue manager by using the call MQDISC(*Hconn*, CompCode, Reason), where *Hconn* is a handle to the queue manager. In the .NET interface, the queue manager is represented by an object of class MQQueueManager. You disconnect from the queue manager by calling the Disconnect() method on that class.

```
// declare an object of type queue manager
MQQueueManager queueManager=new MQQueueManager();
...
// do something...
...
// disconnect from the queue manager
queueManager.Disconnect();
```

Related concepts

Technical overview

“Developing applications for IBM MQ” on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

Related tasks

Contacting IBM Support

Troubleshooting IBM MQ .NET problems

“Developing Microsoft Windows Communication Foundation applications with IBM MQ” on page 1218

The Microsoft Windows Communication Foundation (WCF) custom channel for IBM MQ sends and receives messages between WCF clients and services.

"Developing XMS .NET applications" on page 594
IBM MQ Message Service Client (XMS) for .NET (XMS .NET) provides an application programming interface (API) called XMS that has the same set of interfaces as the Java Message Service (JMS) API. IBM MQ Message Service Client (XMS) for .NET contains a fully managed implementation of XMS, which can be used by any .NET compliant language.

Windows Linux Installing IBM MQ classes for .NET

IBM MQ classes for .NET, including samples, are installed with IBM MQ on Windows and Linux

Prerequisites and installation

V 9.4.0 IBM MQ 9.4.0 provides an IBM MQ .NET client library built against .NET 6 as the target framework. From IBM MQ 9.4.0, Microsoft .NET 6.0 is the minimum required version for running applications using IBM MQ libraries that are built using .NET 6 as the target framework. The IBM MQ .NET client library built using .NET 6 as the target framework is available under *MQ_INSTALLATION_PATH*`/bin` on Windows and under *MQ_INSTALLATION_PATH*`/lib64` on Linux.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for .NET. If you are using a .NET 6 application, you can run this application without any recompilation being required by making a small edit in the `runtimeconfig` file to set the `targetframeworkversion` to `"net8.0"`.

Deprecated V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, in IBM MQ classes for .NET, the methods WriteObject(), ReadObject(), CreateObjectMessage(), and the classes ObjectMessage and XmsObjectMessageImpl used for serialization and deserialization of data are deprecated.

Removed V 9.4.0 V 9.4.0 The IBM MQ .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

The latest version of IBM MQ classes for .NET is installed by default as part of the standard IBM MQ installation in the *Java and .NET Messaging and Web Services* feature.

Windows For more information about prerequisites and installation on Windows:

- See Requirements for IBM MQ classes for .NET for the prerequisite software to run IBM MQ classes for .NET.
- See Installing IBM MQ server on Windows or Installing an IBM MQ client on Windows systems for installation instructions.

Linux For more information about prerequisites and installation on Linux:

- See Requirements for IBM MQ classes for .NET for the prerequisite software to run IBM MQ classes for .NET.
- For rpm installation instructions, see Installing an IBM MQ client on Linux systems.
- For Linux Ubuntu, using Debian packages, see Installing an IBM MQ client on Linux systems.

The IBM MQ classes for .NET Standard library, `amqmdnetstd.dll`, is available for downloading from the NuGet repository. For more information, see "Downloading IBM MQ classes for .NET from the NuGet repository" on page 542.

`amqmdnetstd.dll` library

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, the `amqmdnetstd.dll` library built using .NET 6 as the target framework is available at the following locations:

- Windows On Windows: *MQ_INSTALLATION_PATH*\bin. The sample applications are installed in *MQ_INSTALLATION_PATH*/samp/dotnet/samples/cs/core/base.
- Linux On Linux: *MQ_INSTALLATION_PATH*\lib64. The .NET samples are in *MQ_INSTALLATION_PATH*/samp/dotnet/samples/cs/core/base.

Attention: Removed V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ .NET client libraries built using .NET Standard 2.0 as the target framework are removed. These libraries were deprecated at IBM MQ 9.3.1.

Stabilized LTS The amqmdnet.dll library for .NET Framework is still supplied, but this library is stabilized; that is, no new features will be introduced into it. For any of the latest features you must migrate to the amqmdnetstd.dll library. However, you can continue to use the amqmdnet.dll library on IBM MQ 9.1 or later Long Term Support or Continuous Delivery releases.

V 9.4.0 V 9.4.0 Here are two scenarios that you might encounter following the removal of the netstandard2.0 libraries:

- If you are using a IBM MQ classes for .NET Framework application that is built using the netstandard2.0 libraries such as amqmdnetstd.dll, you need to rebuild your application with the Microsoft.NET Framework 4.7.2 libraries such as amqmdnet.dll, in order for your application to run successfully. If you do not rebuild your application, you might get an System.IO.Unexceptionable message:

```
Exception caught: System.IO.FileLoadException: Could not load file or assembly 'amqmdnetstd,
Version=9.3.5.0, Culture=neutral, PublicKeyToken=23d6cb914eeaac0e' or one of its dependencies.
The located assembly's manifest definition does not match the assembly reference. (Exception
from HRESULT: 0x80131040)
File name: 'amqmdnetstd, Version=9.3.5.0, Culture=neutral, PublicKeyToken=23d6cb914eeaac0e'
   at SimplePut.SimplePut.PutMessages()
   at SimplePut.SimplePut.Main(String[] args) in C:\SampleCode\Program.cs:line 132
```

- If you are using a .NET 6 application that is built using netstandard2.0 libraries, then you just need to replace those libraries with the same .NET 6 libraries in the bin folder of the application runtime directory. No rebuild is required.

  Note: The replacement .NET 6 library should always be of the same or higher level than the replaced netstandard2.0 library.

dspmqver command

You can use the **dspmqver** command to display version and build information for the .NET Core component.

Feature comparison between IBM MQ classes for .NET Framework and IBM MQ classes for .NET

The following table lists the features for IBM MQ classes for .NET Framework compared with the features for IBM MQ classes for .NET

Table 76. Differences between IBM MQ classes for .NET Framework and IBM MQ classes for .NET .

Feature	IBM MQ classes for .NET Framework	IBM MQ classes for .NET
Class Names (APIs)	All classes remain the same in each network.	All classes remain the same in each network.

Table 76. Differences between IBM MQ classes for .NET Framework and IBM MQ classes for .NET . (continued)

Feature	IBM MQ classes for .NET Framework	IBM MQ classes for .NET
Operating System	Windows	Windows Dockerized containers Linux macOS
`app.config` file (Configuration file to enable Trace in redistributable client)	`app.config` file is used to enable trace for the redistributable package and stand-alone IBM MQ .NET client. See Tracing an IBM MQ classes for .NET Framework client using an application configuration file for more information on the variables you use for trace, including **MQTRACEPATH** and **MQTRACELEVEL**.	`app.config` is not supported. Use environment variables.

Table 76. Differences between IBM MQ classes for .NET Framework and IBM MQ classes for .NET . (continued)

Feature	IBM MQ classes for .NET Framework	IBM MQ classes for .NET
Trace	For a full client installation of IBM MQ, you can use the **strmqtrc** command to enable trace for IBM MQ classes for .NET Framework. For redistributable clients, the `app.config` file is also used to enable trace. For more information, see Tracing IBM MQ .NET applications. V 9.4.0 From IBM MQ 9.4.0, you can enable and disable trace by using the `mqclient.ini` file and setting the appropriate properties of the Trace stanza. You can also enable and disable tracing dynamically with the `mqclient.ini` file. For more information, see Tracing IBM MQ .NET applications with mqclient.ini.	The environment variable **MQDOTNET_TRACE_ON** is used to enable trace for redistributable clients. Values equal to and less than 0 do not enable trace. A value of 1 enables default level tracing. A value greater than 1, enables detailed tracing. Setting this environment variable to any other value like string does not enable trace. See Tracing IBM MQ .NET applications using environment variables. The **MQDOTNET_TRACE_ON** environment variable checks whether the IBM MQ trace directory is available or not. If the trace directory is available, the trace file is generated in the trace directory. However, if IBM MQ is not installed, the trace file is copied to the current working directory. Other environment variables including **MQERRORPATH**, **MQLOGLEVEL**, **MQSERVER**, and so on, that are used for IBM MQ classes for .NET Framework, can be used and work in the same way. V 9.4.0 From IBM MQ 9.4.0, you can enable and disable trace by using the `mqclient.ini` file and setting the appropriate properties of the Trace stanza. You can also enable and disable tracing dynamically with the `mqclient.ini` file. For more information, see Tracing IBM MQ .NET applications with mqclient.ini.
Transport Modes	Managed, Unmanaged, and Bindings	Managed

Table 76. Differences between IBM MQ classes for .NET Framework and IBM MQ classes for .NET . (continued)

Feature	IBM MQ classes for .NET Framework	IBM MQ classes for .NET
TLS	The Windows keystore is used for storing the certificates.	Windows On Windows, the keystore must be used for storing the certificates. Permitted values are *USER or *SYSTEM. Based on the input, the IBM MQ .NET client looks at the Windows key store of the current user, or System wide. Linux On Linux, it is recommended to use the X509Store class to install certificates and .NET Core installs certificates to the following location: `".dotnet/corefx/cryptography/x509stores"`.
CCDT	Supported	Supported, and the settings of the CCDT path are the same as for .NET Framework classes.
Client auto reconnect	Supported	Supported
Distributed transactions	Supported	Not supported
Installation of dynamic linked libraries (dll's) into the global assembly cache (GAC)	Dll's are installed into the GAC as part of the IBM MQ installation.	Dll's are not installed into the GAC as part of the IBM MQ installation.

Note: Windows Windows security identifiers (SIDs):

Domain level authentication is not supported for IBM MQ classes for .NET (.NET Standard and .NET 6 libraries). The logged-in user ID is used for authentication.

Developing IBM MQ .NET Core applications on macOS

macOS

IBM MQ .NET Core applications can be developed on macOS.

The IBM MQ .NET libraries are not packaged with the macOS toolkit so you must copy them from a Windows or Linux IBM MQ client on to macOS. You can then use these libraries to develop IBM MQ .NET Core applications on macOS.

Once developed, these applications can be run supported on either Windows or Linux environments.

Related concepts

“Installing IBM MQ classes for .NET Framework” on page 543
IBM MQ classes for .NET Framework, including samples, are installed with IBM MQ. There is a prerequisite of Microsoft.NET Framework on Windows.

“Installing IBM MQ classes for XMS .NET” on page 599

IBM MQ classes for XMS .NET, including samples, are installed with IBM MQ on Windows and Linux.

Windows Linux Downloading IBM MQ classes for .NET from the NuGet repository

The IBM MQ classes for .NET are available for downloading from the NuGet repository, so that they can be easily consumed by .NET Developers.

About this task

NuGet is the package manager for Microsoft development platforms including .NET. The NuGet client tools provide the ability to produce and consume packages. A NuGet package is a single compressed file with the `.nupkg` extension that contains compiled code (DLLs), other files related to that code, and a descriptive manifest that includes information like the package's version number.

You can download the `IBMMQDotnetClient` NuGet package, which contains the `amqmdnetstd.dll` library, from the NuGet Gallery, which is is the central package repository used by all package authors and consumers.

Note: V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, the NuGet package contains libraries built using .NET 6 as the target framework.

Removed The IBM MQ .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for .NET. If you are using a .NET 6 application, you can run this application without any recompilation being required by making a small edit in the `runtimeconfig` file to set the `targetframeworkversion` to `"net8.0"`.

There are three ways of downloading the `IBMMQDotnetClient` package:

- By using Microsoft Visual Studio. NuGet is distributed as a Microsoft Visual Studio extension. From Microsoft Visual Studio 2012, NuGet is pre-installed by default.
- From the command line using either the NuGet Package Manager or the .NET CLI.
- By using a web browser.

As for the redistributable package, you enable trace by using the environment variable **`MQDOTNET_TRACE_ON`**.

Procedure

- To download the `IBMMQDotnetClient` package by using the Package Manager UI in Microsoft Visual Studio, complete the following steps:
  a) Right-click the .NET project and then click **Manage Nuget Packages**.
  b) Click the **Browse** tab and search for `"IBMMQDotnetClient"`.
  c) Select the package and click **Install**.
  During installation, the Package Manager provides progress information in the form of console statements.
- To download the `IBMMQDotnetClient` package from the command line, choose one of the following options:
  - Using the NuGet Package Manager, enter the following command:

    ```
    Install-Package IBMMQDotnetClient -Version 9.1.4.0
    ```

    During installation, the Package Manager provides progress information in the form of console statements. You can redirect the output to a log file.

- Using the .NET CLI, enter the following command:

```
dotnet add package IBMMQDotnetClient --version 9.1.4
```

- Using a web browser, download the `IBMMQDotnetClient` package from https://www.nuget.org/packages/IBMMQDotnetClient.

Related concepts

IBM MQ Client for .NET license information

Related tasks

"Downloading IBM MQ classes for XMS .NET from the NuGet repository" on page 602
The IBM MQ classes for XMS .NET are available for downloading from the NuGet repository, so that they can be easily consumed by .NET Developers.

Windows Installing IBM MQ classes for .NET Framework

IBM MQ classes for .NET Framework, including samples, are installed with IBM MQ. There is a prerequisite of Microsoft.NET Framework on Windows.

The latest version of IBM MQ classes for .NET Framework is installed by default as part of the standard IBM MQ installation in the *Java and .NET Messaging and Web Services* feature. For installation instructions, see Installing IBM MQ server on Windows or Installing an IBM MQ client on Windows systems.

From IBM MQ 9.3.0, to run IBM MQ classes for .NET Framework you must install Microsoft.NET Framework V4.7.2 or later.

Existing applications that are compiled with Microsoft.NET Framework V3.5 can be run without recompiling by adding the following tag in the `app.config` file of the application:

```
<configuration>
 <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
 </startup>
</configuration>
```

Note: If the Microsoft .NET Framework V4.7.2 or higher is not installed before you install IBM MQ, then the IBM MQ product installation continues without error, but the IBM MQ classes for .NET is not available. If the.NET Framework is installed after you install IBM MQ, then the IBM MQ.NET assemblies must be registered by running the *`WMQInstallDir`*`\bin\amqiRegisterdotNet.cmd` script, where *WMQInstallDir* is the directory where IBM MQ is installed. This script installs the required assemblies in the Global Assembly Cache (GAC). A set of `amqi*.log` files that record the actions that are taken are created in the %TEMP% directory. It is not necessary to rerun the `amqiRegisterdotNet.cmd` script if .NET is upgraded to V4.7.2 or higher from an earlier version, for example, from .NET V3.5.

In a multiple installation environment, if you previously installed the IBM MQ classes for .NET as a support pack, you cannot install IBM MQ unless you first uninstall the support pack. The IBM MQ classes for .NET feature that is installed with IBM MQ contains the same functions as the support pack.

Sample applications, including source files, are also supplied; see "Sample applications for .NET" on page 544.

For information about using the IBM MQ custom channel for the Microsoft WCF with .NET, see "Developing Microsoft Windows Communication Foundation applications with IBM MQ" on page 1218

Related concepts

"Installing IBM MQ classes for .NET" on page 537
IBM MQ classes for .NET, including samples, are installed with IBM MQ on Windows and Linux

Related tasks

Tracing IBM MQ .NET applications

Options for connecting IBM MQ classes for .NET to a queue manager

There are three modes of connecting IBM MQ classes for .NET to a queue manager. Consider which type of connection best suits your requirements.

Client bindings connection

To use IBM MQ classes for .NET as an IBM MQ MQI client, you can install it, with the IBM MQ MQI client, either on the IBM MQ server machine, or on a separate machine. A client bindings connection can use XA or non-XA transactions

Server bindings connection

When used in server bindings mode, IBM MQ classes for .NET use the queue manager API, rather than communicating through a network. This provides better performance for IBM MQ applications than using network connections.

To use the bindings connection, you must install IBM MQ classes for .NET on the IBM MQ server.

Managed client connection

A connection made in this mode connects as an IBM MQ client to an IBM MQ server running either on the local or a remote machine.

The IBM MQ classes for .NET connecting in this mode remain in .NET managed code and make no calls to native services. For more information about managed code, refer to Microsoft documentation.

There are a number of limitations to using the managed client. For more information about these, see “Managed client connections” on page 559.

Sample applications for .NET

To run your own .NET applications, use the instructions for the verification programs, substituting your application name in place of the sample applications.

The following sample applications are supplied:

- A put message application
- A get message application
- A 'hello world' application
- A publish/subscribe application
- An application using message properties

All these sample applications are supplied in the C# language, and some are also supplied in C++ and Visual Basic. You can write applications in any language supported by .NET.

"Put message" program SPUT (nmqsput.cs, mmqsput.cpp, vmqsput.vb)
This program shows how to put a message to a named queue. The program has three parameters:

- The name of a queue (required), for example, SYSTEM.DEFAULT.LOCAL.QUEUE
- The name of a queue manager (optional)
- The definition of a channel (optional), for example, SYSTEM.DEF.SVRCONN/TCP/hostname(1414)

If no queue manager name is given, the queue manager defaults to the default local queue manager. If a channel is defined, it has the same format as the MQSERVER environment variable.

"Get message" program SGET (nmqsget.cs, mmqsget.cpp, vmqsget.vb)
This program shows how to get a message from a named queue. The program has three parameters:

- The name of a queue (required), for example, SYSTEM.DEFAULT.LOCAL.QUEUE
- The name of a queue manager (optional)

- The definition of a channel (optional), for example, SYSTEM.DEF.SVRCONN/TCP/hostname(1414)

If no queue manager name is given, the queue manager defaults to the default local queue manager. If a channel is defined, it has the same format as the MQSERVER environment variable.

"Hello World" program (nmqwrld.cs, mmqwrld.cpp, vmqwrld.vb)
This program shows how to put and get a message. The program has three parameters:

- The name of a queue (optional), for example, SYSTEM.DEFAULT.LOCAL.QUEUE or SYSTEM.DEFAULT.MODEL.QUEUE
- The name of a queue manager (optional)
- A channel definition (optional), for example, SYSTEM.DEF.SVRCONN/TCP/hostname(1414)

If no queue name is given, the name defaults to SYSTEM.DEFAULT.LOCAL.QUEUE. If no queue manager name is given, the queue manager defaults to the default local queue manager.

"Publish/subscribe" program (MQPubSubSample.cs)
This program shows how to use IBM MQ publish/subscribe. It is supplied in C# only. The program has two parameters:

- The name of a queue manager (optional)
- A channel definition (optional)

"Message properties" program (MQMessagePropertiesSample.cs)
This program shows how to use message properties. It is supplied in C# only. The program has two parameters:

- The name of a queue manager (optional)
- A channel definition (optional)

You can verify your installation by compiling and running these applications.

Installation locations

The sample applications are installed to the following locations, according to the language in which they are written. *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

C#

MQ_INSTALLATION_PATH\Tools\dotnet\samples\cs\nmqswrld.cs

MQ_INSTALLATION_PATH\Tools\dotnet\samples\cs\nmqsput.cs

MQ_INSTALLATION_PATH\Tools\dotnet\samples\cs\nmqsget.cs

MQ_INSTALLATION_PATH\Tools\dotnet\samples\cs\MQPubSubSample.cs

MQ_INSTALLATION_PATH\Tools\dotnet\samples\cs\MQMessagePropertiesSample.cs

Managed C++

MQ_INSTALLATION_PATH\Tools\dotnet\samples\mcp\mmqswrld.cpp

MQ_INSTALLATION_PATH\Tools\dotnet\samples\mcp\mmqsput.cpp

MQ_INSTALLATION_PATH\Tools\dotnet\samples\mcp\mmqsget.cpp

Visual Basic

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\vmqswrld.vb

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\vmqsput.vb

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\vmqsget.vb

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\xmqswrld.vb

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\xmqsput.vb

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\xmqsget.vb

Building the sample applications

To build the sample applications, a batch file is supplied for each language.

C#

MQ_INSTALLATION_PATH\Tools\dotnet\samples\cs\bldcssamp.bat

The bldcssamp.bat file contains a line for each sample, which is all that is necessary to build this sample program:

```
csc /t:exe /r:System.dll /r:amqmdnet.dll /lib: MQ_INSTALLATION_PATH\bin
/out:nmqwrld.exe nmqwrld.cs
```

Managed C++

MQ_INSTALLATION_PATH\Tools\dotnet\samples\mcp\bldmcpsamp.bat

The bldmcpsamp.bat file contains a line for each sample, which is all that is necessary to build this sample program:

```
cl /clr:oldsyntax MQ_INSTALLATION_PATH\bin mmqwrld.cpp
```

If you want to compile these applications on Microsoft Visual Studio 2003/.NET SDKv1.1, replace the compile command:

```
cl /clr:oldsyntax MQ_INSTALLATION_PATH\bin mmqwrld.cpp
```

with

```
cl /clr MQ_INSTALLATION_PATH\bin mmqwrld.cpp
```

Visual Basic

MQ_INSTALLATION_PATH\Tools\dotnet\samples\vb\bldvbsamp.bat

The bldvbsamp.bat file contains a line for each sample, which is all that is necessary to build this sample program:

```
vbc /r:System.dll /r: MQ_INSTALLATION_PATH\bin\amqmdnet.dll /out:vmqwrld.exe vmqwrld.vb
```

Samples for using IBM MQ with Microsoft .NET Core

IBM MQ supports .NET Core for IBM MQ .NET applications in Windows environments. IBM MQ classes for .NET Standard, including samples, are installed by default as part of the standard IBM MQ installation.

The sample applications for IBM MQ .NET are installed in `&MQINSTALL_PATH&/samp/dotnet/samples/cs/core/base`. A script is also provided, which can be used to compile the samples.

You can build the samples by using the supplied `build.bat` files. There is one `build.bat` for each sample in the following location on Windows:

- `MQ\tools\dotnet\samples\cs\core\base\SimpleGet`
- `MQ\tools\dotnet\samples\cs\core\base\SimplePut`

Linux IBM MQ also supports Core for applications in Linux environments.

For more infromation about using IBM MQ with Microsoft .NET Core, see “Installing IBM MQ classes for .NET” on page 537.

Configuring your queue manager to accept TCP/IP client connections

Configure a queue manager to accept incoming connection requests from the clients.

About this task

This task explains the basic steps for configuring a queue manager to accept TCP/IP client connections. For a production system, you must also consider the security implications when configuring queue managers.

Procedure

1. Define a server connection channel:
   a. Start the queue manager.
   b. Define a sample channel called NET.CHANNEL:

```
DEF CHL('NET.CHANNEL') CHLTYPE(SVRCONN) TRPTYPE(TCP) MCAUSER(' ') +
DESCR('Sample channel for IBM MQ classes for .NET')
```

   Important: This sample is intended for use in a sandbox environment only, as it does not include any consideration of security implications. For a production system, consider using TLS or a security exit. See Securing IBM MQ for more information.
2. Start a listener:

```
runmqlsr -t tcp [-m qmnqme ] [-p portnum ]
```

   Note: The square brackets indicate optional parameters; *qmname* is not required for the default queue manager, and the port number *portnum* is not required if you are using the default (1414).

Distributed transactions in .NET

Distributed transactions or global transactions allows client applications to include several different sources of data on two or more networked systems in one transaction.

In distributed transactions, a transaction manager coordinates and manages the transaction among two or more resource managers.

Transactions can be either single phase or two-phase commit process. The single-phase commit is a process where only one resource manager participates in the transaction and two-phase commit process is where there are more than one resource manager participating in the transaction. In the two-phase commit process, the transaction manager sends a prepare call to check whether all the resource managers are prepared to commit. When it receives the acknowledgment from all the resource managers, the commit call is issued. Else, a rollback on the whole transaction happens. See Transaction management and support for more details. The resource managers should inform the transaction managers of their participation in the transaction. When the resource manager informs the transaction manager of its participation, the resource manager gets callbacks from the transaction manager when the transaction is going to commit or roll back.

IBM MQ .NET classes already supports distributed transactions in unmanaged and server bindings mode connections. In these modes, IBM MQ .NET classes delegates all its calls to C extended transaction client, which manages the transaction processing on behalf of .NET.

IBM MQ.NET classes now support distributed transactions in managed mode where IBM MQ .NET Classes uses System.Transactions namespace for the distributed transactions support. The System.Transactions infrastructure makes transactional programming simple and efficient by supporting the transactions initiated in all the resource managers including IBM MQ. The IBM MQ .NET application can put and get messages using .NET implicit transaction programming or explicit transaction programming model. In implicit transactions, the transaction boundaries are created by the application program that decides

when to commit, rollback (for explicit transactions) or complete the transaction. In explicit transactions, you have to explicitly specify whether you want to commit, roll back, and complete the transaction.

IBM MQ.NET uses Microsoft distributed transaction coordinator (MS DTC) as the transaction manager, which coordinates and manages the transaction between multiple resource managers. IBM MQ is used as the resource manager. Note that you cannot use TLS with XA transactions. You must use CCDT. For more information, see Using the extended transactional client with TLS channels.

IBM MQ.NET follows the X/Open Distributed Transaction Processing (DTP) model. The X/Open Distributed Transaction Processing model is a distributed transaction processing model proposed by the Open Group, a vendor consortium. This model is a standard among most of the commercial vendors in the transaction processing and database domains. Most of the commercial transaction management products support the X/DTP model.

Modes of transaction

- “Distributed transactions in .NET managed mode” on page 549
- Distributed transactions for unmanaged mode

Coordinating transactions in various scenarios

- A connection might participate in several transactions, but only one transaction is active at any point of time.
- During a transaction, the MQQueueManager.Disconnect call is honored. In this case the transaction is asked to roll back.
- During a transaction, the MQQueue.Close or MQTopic.Close call is honored. In this case transaction is asked to roll back.
- The transaction boundaries are created by the application program that decides when to commit, rollback (for explicit transactions) or complete (for implicit transactions) the transaction.
- If the client application breaks during a transaction with an unexpected error before issuing a Put or Get call on a queue or topic call, the transaction is rolled back and a MQException is thrown.
- If MQCC_FAILED reason code is returned during a Put or Get call on a queue or Topic call, an MQException is thrown with reason code and the transaction is rolled. If a prepare call has been already issued by the transaction manager, then IBM MQ .NET returns the prepare request by forcibly rolling back the transaction. Then the transaction manager DTC causes a rollback on current work with all the resource managers in current ambient transactions.
- During a transaction involving multiple resource managers if some environmental reason causes the Put or Get call to hang indefinitely, the transaction manager waits until a stipulated time. After the time is out, it causes the rollback of all current work with all the resource managers in current ambient transactions. If this indefinite wait happens during the prepare phase, the transaction manager might timeout or issue an in-doubt call on the resource in which case the transaction is rolled back.
- Applications using transactions must Put or Get messages under SYNC_POINT. If a message Put or Get call is issued under a transactional context that is not under SYNC_POINT, the call fails with MQRC_UNIT_OF_WORK_NOT_STARTED reason code.

Behavioral differences between Managed and Unmanaged Client transaction support using Microsoft.NET System.Transactions namespace

Nested Transactions have a TransactionScope inside another TransactionScope

- IBM MQ .NET fully managed client does support nested TransactionScope
- IBM MQ .NET unmanaged client does not support nested TransactionScope

Dependent Transactions from System.Transactions

- IBM MQ .NET fully managed client does support the dependent transactions facility provided by System.Transactions.

- IBM MQ .NET unmanaged client does not support the dependent transactions facility provided by System.Transactions.

Product samples

Product samples `SimpleXAPut`, and `SimpleXAGet` are available under `WebSphere MQ\tools\dotnet\samples\cs\base`. The samples are C# applications, which demonstrate using MQPUT and MQGET under Distributed Transactions using SystemTransactions namespace. For more information about these samples, see "Creating simple put and get messages within a TransactionScope" on page 552.

Distributed transactions in .NET managed mode

IBM MQ .NET classes use System.Transactions namespace for the distributed transactions support in managed mode. In the managed mode, MS DTC coordinates and manages distributed transactions across all the servers enlisted in a transaction.

IBM MQ .NET classes provide an explicit programming model based on the System.Transactions.Transaction class and an implicit programming model using the System.Transactions.TransactionScope, class where the transactions are automatically managed by the infrastructure.

Implicit Transaction

The following piece of code describes how an IBM MQ .NET application puts a message using .NET implicit transaction programming.

```
Using (TransactionScope scope = new TransactionScope ())
{
    Q.Put (putMsg,pmo);
    scope.Complete ();
}

Q.close();
qMgr.Disconect();}
```

Explanation of the code flow of implicit transaction

The code creates *TransactionScope* and puts the message under the scope. It then calls *Complete* to inform the transaction coordinator of the completion of the transaction. The transaction coordinator now issues *prepare* and *commit* to complete the transaction. If an issue is detected, then a *rollback* is called.

Explicit Transaction

The following code describes how an IBM MQ .NET application puts messages using .NET explicit transaction programming model.

```
MQQueueManager qMgr = new MQQueuemanager ("MQQM);
MQQueue Q = QMGR.AccessQueue("Q", MQC.MQOO_OUTPUT+MQC.MQOO_INPUT_SHARED);
MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.Options = MQC.MQPMO_SYNCPOINT;
MQMessage putMsg1 = new MQMessage();
Using(CommittableTransaction tx = new CommittableTransaction()){
Transaction.Current = tx;
    try
    {
    Q.Put(MSG,pmo);
    tx.commit();
    }
    catch(Exception)
    {tx.rollback();}
    }

Q.close();
qMgr.Disconnect();
}
```

Explanation of the code flow of explicit transaction

The piece of code creates transaction using *CommitableTransaction* class. It puts a message under that scope and then explicitly calls *commit* to complete the transaction. If there are any issues *rollback* is called.

Distributed transactions in .NET unmanaged mode

IBM MQ.NET classes support unmanaged connections (client) using extended transaction client and COM+/MTS as the transaction coordinator, using either implicit or explicit transaction programming model. In the unmanaged mode, IBM MQ .NET classes delegate all its calls to C extended transaction client that manages the transaction processing on behalf of .NET.

The transaction processing is controlled by an external transaction manager, coordinating the global unit of work under the control of the API of the transaction manager. The MQBEGIN, MQCMIT, and MQBACK verbs are unavailable. IBM MQ .NET classes expose this support by way of its unmanaged transport mode (C client). See Configuring XA-compliant transaction managers

MTS is evolved as a transaction processing (TP) system to provide the same features on Windows NT as available in CICS, Tuxedo, and on other platforms. When the MTS is installed, a separate service is added to Windows NT called the Microsoft Distributed Transaction Coordinator (MSDTC). The MSDTC coordinates the transactions that span separate data stores or resources. To work, it requires each data store to implement its own proprietary resource manager.

IBM MQ becomes compatible with MSDTC by implementing an interface (proprietary resource manager interface) where it manages to map DTC XA calls to IBM MQ(X/Open) calls. IBM MQ plays the role of a resource manager.

When a component such as COM+ requests access to an IBM MQ, the COM usually checks with the appropriate MTS context object if a transaction is required. If a transaction is required, the COM informs the DTC and automatically starts an integral IBM MQ transaction for this operation. Then the COM works with the data through the MQMTS software, putting and getting messages as required. The object instance obtained from the COM calls the SetComplete or SetAbort method after all the actions on the data are over. When the application issues SetComplete, the call signals the DTC that the application has completed the transaction and the DTC can go ahead with the two-phase commit process. The DTC then issues calls to MQMTS which in turn issues calls to IBM MQ to commit or roll back the transaction.

Writing an IBM MQ .NET application using unmanaged client

To run within the context of COM+, a .NET class must inherit from System .EnterpriseServices.ServicedComponent. The rules and recommendations to create assemblies that use serviced components are the following:

Note: The following steps are relevant only if you are using System.EnterpriseServices mode.

- The class and method being started in COM+ must both be public (no internal classes, and no protected or static methods).
- The class and method attributes: The TransactionOption attribute dictates the transaction level of the class, that is whether the transactions are disabled, supported, or required. The AutoComplete attribute on the ExecuteUOW() method instructs COM+ to commit the transaction if no unhandled exception is thrown.
- Strong-naming an assembly: The assembly must be strong-named and registered in the Global Assembly Cache (GAC). The assembly is registered in COM+ explicitly or by lazy registration after it is registered in the GAC.
- Registering an assembly in COM+: Prepare the assembly to be exposed to COM clients. Then create a type library by using the Assembly Registration tool, regasm.exe.

```
regasm UnmanagedToManagedXa.dll
```

- Register the assembly into GAC `gacutil /i UnmanagedToManagedXa.dll`.

- Register the assembly in COM+ by using the .NET services installer tool, regsvcs.exe. See the type library created by regasm.exe:

```
Regsvcs /appname:UnmanagedToManagedXa /tlb:UnmanagedToManagedXa.tlb  UnmanagedToManagedXa.dll
```

- The assembly is deployed into the GAC, and later it is registered in COM+ by lazy registration. The .NET framework takes care of the registration after the code is run for the first time.

The example code flow using System.EnterpriseServices model and System.Transactions with COM+ are described in the following sections:

Example code flow using System.EnterpriseServices model

```
using System;
using IBM.WMQ;
using IBM.WMQ.Nmqi;
using System.Transactions;
using System.EnterpriseServices;

namespace UnmanagedToManagedXa
{

[ComVisible(true)]
[System.EnterpriseServices.Transaction(System.EnterpriseServices.TransactionOption.Required)]
    public class MyXa : System.EnterpriseServices.ServicedComponent
    {

        public MQQueueManager QMGR = null;
        public MQQueueManager QMGR1 = null;
        public MQQueue QUEUE = null;
        public MQQueue QUEUE1 = null;
        public MQPutMessageOptions pmo = null;
        public MQMessage MSG = null;

        public MyXa()
        {
        }

        [System.EnterpriseServices.AutoComplete()]
        public void ExecuteUOW()
        {
            QMGR = new MQQueueManager("usemq");

            QUEUE = QMGR.AccessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
                                     MQC.MQOO_INPUT_SHARED +
                                     MQC.MQOO_OUTPUT +
                                     MQC.MQOO_BROWSE);

            pmo = new MQPutMessageOptions();
            pmo.Options = MQC.MQPMO_SYNCPOINT;
            MSG = new MQMessage();
            QUEUE.Put(MSG, pmo);
            QMGR.Disconnect();
        }
    }
}


public void RunNow()
{
   MyXa xa = new MyXa();
   xa.ExecuteUOW();
}
```

Example code flow using System.Transactions for interactions with COM+

```
[STAThread]
public void ExecuteUOW()
{
Hashtable t1 = new Hashtable();
t1.Add(MQC.CHANNEL_PROPERTY, "SYSTEM.DEF.SVRCONN");
t1.Add(MQC.HOST_NAME_PROPERTY, "localhost");
t1.Add(MQC.PORT_PROPERTY, 1414);
t1.Add(MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES_CLIENT);
TransactionOptions opts = new TransactionOptions();

using(TransactionScope scope = new  TransactionScope(TransactionScopeOption.RequiresNew,
                                 opts,     EnterpriseServicesInteropOption.Full)
  {

        QMGR = new MQQueueManager("usemq", t1);
        QUEUE = QMGR.AccessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE",
```

```
                                          MQC.MQOO_INPUT_SHARED +
                                          MQC.MQOO_OUTPUT +
                                          MQC.MQOO_BROWSE);

          pmo = new MQPutMessageOptions();
          pmo.Options = MQC.MQPMO_SYNCPOINT;
          MSG = new MQMessage();
          QUEUE.Put(MSG, pmo);
          scope.Complete();
   }
  QMGR.Disconnect();
}
```

Creating simple put and get messages within a TransactionScope

Product sample C# applications are available within IBM MQ. These simple applications demonstrate putting and getting messages within a TransactionScope. At the end of the task, you will be able to put and get messages from a queue or topic.

Before you begin

MSDTC service must be running and enabled for XA Transactions.

About this task

The example is a simple application, `SimpleXAPut` and `SimpleXAGet`. The programs `SimpleXAPut` and `SimpleXAGet` are C# applications available within IBM MQ. `SimpleXAPut` demonstrates using MQPUT, under Distributed Transactions using SystemTransactions namespace. `SimpleXAGet` demonstrates using MQGET, under Distributed Transactions using SystemTransactions namespace.

`SimpleXAPut` is located in `MQ\tools\dotnet\samples\cs\base`

Procedure

The applications can be run with the command-line parameters from `tools\dotnet\samples\cs\base\bin`

```
SimpleXAPut.exe -d destinationURI [-h host -p port -l channel -tx transaction -tm mode -n
numberOfMsgs]
```

```
SimpleXAGet.exe -d destinationURI [-h host -p port -l channel -tx transaction -tm mode -n
numberOfMsgs]
```

where the parameters are:

`-destinationURI`

This can be queue or topic. For a queue, specify as `queue://queueName` and for a topic specify as `topic://topicName`.

`-host`

This can be a host name such as localhost or an IP address.

`-port`

The port on which the queue manager is running.

`-channel`

The connection channel being used. The default is `SYSTEM.DEF.SVRCONN`

`-transaction`

The transaction result, for example commit or rollback.

-mode

The transport mode, for example managed or unmanaged.

-numberOfMsgs

The number of messages. The default is 1.

Example

```
SimpleXAPut -d topic://T01 -h localhost -p 2345 -tx rollback -tm unmanaged
```

```
SimpleXAGet -d queue://Q01 -h localhost -p 2345 -tx rollback -tm unmanaged
```

Recovering transactions in IBM MQ .NET

This section describes the process of recovering transactions in IBM MQ .NET XA using managed mode.

About this task

In distributed transaction processing, the transactions can be successfully completed, but there can be scenarios where a transaction might fail for many reasons. These reasons might include a system failure, hardware failure, network error, incorrect or invalid data, application errors or natural or man-made disasters. It is not possible to prevent transaction failures. The distributed transaction system must be capable of handling these failures. It must be able to detect and correct errors when they occur. This process is known as Transaction Recovery.

An important aspect of the Distributed Transaction Processing is to recover the incomplete or in doubt transactions. It is essential to run the recovery as the Unit of Work part of a particular transaction is held locked until it is recovered. Microsoft.NET from its System.Transactions class library provides the option for recovering incomplete/in-doubt transactions. This recovery support expects Resource Manager to maintain the transaction logs and run the recovery when in need.

In the Microsoft .NET transaction recovery model, the Transaction Manager (System.Transactions, or Microsoft Distributed Transaction coordinator (MS DTC), or both), initiates, coordinates, and controls the transaction recovery. The OLE Tx Protocol (the Microsoft XA protocol) based Resource Managers provide the options to configure the DTC to drive, coordinate, and control the recovery for them. To do this, Resource Managers must register XA_Switch with MS DTC by using native interface.

XA_Switch provides the entry points of XA functions like xa_start, xa_end, and xa_recover in the Resource Manager to the Distributed Transaction Coordinator.

Recovery using Microsoft Distributed Transaction coordinator (DTC):

Microsoft Distributed Transaction coordinator provides two kinds of recovery processes.

Cold Recovery

Cold recovery is performed if the transaction manager process fails while a connection to a XA resource manager is open. When the transaction manager restarts, it reads the transaction manager logs and re-establishes the connection to the XA resource manager and then initiates recovery.

Hot Recovery

Hot recovery is performed if the transaction manager remains up while the connection between the transaction manager and the XA resource manager fails because the XA resource manager or the network fails. After the failure, the transaction manager periodically attempts to reconnect to the XA resource manager. When the connection is re-established, the transaction manager initiates XA recovery.

System.Transactions namespace provides managed implementation of Distributed transactions that are based on MS DTC as the transaction manager. It provides similar features as that of MS DTC's

native interface but in fully managed environment. The only difference is about the transaction recovery. System.Transactions expects Resource Managers to drive the recovery by themselves and then coordinate with the Transaction Managers (MS DTC). Resource Manager must ask for recovery of a particular incomplete transaction and then Transaction Manager accepts it and coordinates based on the actual outcome of that particular transaction.

Transaction recovery process for IBM MQ .NET

This section describes how distributed transactions can be recovered with IBM MQ .NET classes.

Overview

To recover an incomplete transaction, the recovery information is required. The transaction recovery information must be logged to storage by the resource managers. IBM MQ .NET classes follow a similar path. The transaction recovery information is logged to a system queue called SYSTEM.DOTNET.XARECOVERY.QUEUE.

Transaction recovery in IBM MQ .NET is a two stage process:

1. Logging of transaction recovery information in SYSTEM.DOTNET.XARECOVERY.QUEUE.
2. Recovering transactions by using the XA Monitor application WmqDotnetXAMonitor.

SYSTEM.DOTNET.XARECOVERY.QUEUE

SYSTEM.DOTNET.XARECOVERY.QUEUE is a system queue that holds transaction recovery information for incomplete transactions. This queue is created when a queue manager is created.

For every transaction, during the prepare phase a persistent message containing the recovery information is added to SYSTEM.DOTNET.XARECOVERY.QUEUE. The message is deleted if the commit call succeeds.

Note: You must not delete the SYSTEM.DOTNET.XARECOVERY.QUEUE queue.

WMQDotnetXAMonitor application

IBM MQ .NET XA Monitor application, WmqDotnetXAMonitor, is a .NET managed application that monitors a queue manager, processes messages in SYSTEM.DOTNET.XARECOVERY.QUEUE and recovers incomplete transactions

If the message channel agent (MCA) is unable to put the message to the destination queue, it generates an exception report containing the original message, and puts it on a transmission queue to be sent to the reply-to queue specified in the original message. (If the reply-to queue is on the same queue manager as the MCA, the message is put directly to that queue, not to a transmission queue.)

The following are deemed to be incomplete transactions and are recovered:

- If the transaction is prepared but COMMIT did not complete within the timeout period.
- If the transaction is prepared but IBM MQ queue manager has gone down.
- If the transaction is prepared but then Transaction Manager has gone down.

  The XA Monitor application must be run from the same system where your IBM MQ .NET client application is running. If there are applications that are running on multiple systems and connecting to the same queue manager, WmqDotnetXAMonitor application must be run from all the systems. Though each client machine has an instance of the XA Monitor application running to recover the application, each XA Monitor instance should be able to identify the message that corresponds to transaction that the current XA Monitor's local MS DTC was coordinating so that it can reenlist and complete it.

Related concepts

“Transaction recovery use cases for IBM MQ .NET” on page 555
There are several different use cases from which transactions might need to be recovered.

Related tasks

“Using the WMQDotnetXAMonitor application” on page 556

The IBM MQ .NET client provides an XA Monitor application, WmqDotnetXAMonitor, that you can use to recover any incomplete distributed transactions. The WmqDotnetXAMonitor application establishes a connection to the queue manager where the transactions are in-doubt and then resolves the transaction based on the parameters that you set.

Transaction recovery use cases for IBM MQ .NET

There are several different use cases from which transactions might need to be recovered.

- **IBM MQ Application using single DTC and single queue manager instance:** In this use case, when you connect to the queue manager and run Unit of Work (UoW) under transaction, and if the transaction fails and becomes incomplete, the XA Monitor application recovers the transaction and completes it.

  In this use case, there will be a single instance of the XA Monitor application running, as a single queue manager is associated with the transactions.
- **Multiple IBM MQ applications using single DTC and single queue manager instance:** In this use case, there are more than one IBM MQ application under single DTC and all are connecting to the same queue manager and running UoW under transactions.

  If the transactions fail and become incomplete, the XA Monitor application recovers them and completes the transactions pertaining to all the applications.

  In this use case, a single instance of the XA Monitor application runs, as one queue manager is used in transactions.
- **Multiple IBM MQ Applications, multiple DTCs, different queue manager instances:** In this use case, there are more than one IBM MQ application under different DTCs (that is, each application is running on a different machine) and connecting to different queue managers.

  If failure occurs and transaction becomes incomplete, monitor application checks the TransactionManagerWhereabouts in the message to determine the DTC address. If the TransactionManagerWhereabouts value matches with the DTC address under which the monitor is running, it completes the recovery, else, it continues to search until the message corresponding to its DTC is found.

  In this use case, there will be only one instance of the XA Monitor application running per client (user or computer) as each client has its own queue manager used in transactions.
- **Multiple IBM MQ applications, multiple DTCs, multiple same queue manager instances:** In this use case, there are more than one IBM MQ application under different DTCs (that is each application is running on a different machine) and all are connecting to same queue manager.

  If failure occurs and transaction becomes incomplete, monitor application verifies the TransactionManagerWhereabouts in the message to check if the DTC address and value match with the DTC under which the monitor is running. If both the values match, it completes the recovery else continues to search until it finds the message corresponding to its DTC.

  In this use case, there will be only a single instance of the XA Monitor application running per client (user or computer), as each client has its own queue manager association used in transactions.
- **Multiple IBM MQ Applications, single DTC, different queue manager instances:** In this use case, there are more than one IBM MQ applications under a single DTC (that is, on a computer, there are more than one IBM MQ applications running) and connecting to different queue managers.

  If transaction fails and becomes incomplete, monitor application recovers the transaction.

  In this use case, there will be as many instances of monitor application running as queue managers connected to, as each application has its own queue manager used in transactions and each of it must be recovered.

  Note: If the XA monitor application is not running in the background, you can start it.

Related concepts

"Transaction recovery process for IBM MQ .NET" on page 554

This section describes how distributed transactions can be recovered with IBM MQ .NET classes.

Related tasks

"Using the WMQDotnetXAMonitor application" on page 556

The IBM MQ .NET client provides an XA Monitor application, WmqDotnetXAMonitor, that you can use to recover any incomplete distributed transactions. The WmqDotnetXAMonitor application establishes a connection to the queue manager where the transactions are in-doubt and then resolves the transaction based on the parameters that you set.

Using the WMQDotnetXAMonitor application

The IBM MQ .NET client provides an XA Monitor application, WmqDotnetXAMonitor, that you can use to recover any incomplete distributed transactions. The WmqDotnetXAMonitor application establishes a connection to the queue manager where the transactions are in-doubt and then resolves the transaction based on the parameters that you set.

About this task

The WMQDotnetXAMonitor application must be run manually. It can be started at any time. You can start it when you see the messages on the SYSTEM.DOTNET.XARECOVERY.QUEUE or you can keep it running in the background before you do any transactional work with the applications that are written using IBM MQ .NET classes.

You can either set the parameter values for WMQDotnetXAMonitor through the command line or by using an application configuration file. Values that are provided through the application configuration file take precedence over values set through the command line.

Before IBM MQ 9.3.0, the connection that WMQDotnetXAMonitor establishes is a nonsecure connection.

From IBM MQ 9.3.0, you have the option of establishing a secure connection to the queue manager by setting additional parameters for WMQDotnetXAMonitor.

Procedure

- To provide input to WmqDotNETXAMonitor by using an application configuration file, see "WmqDotNETXAMonitor application configuration file settings" on page 558.
- To start the WMQDotnetXAMonitor application from the command line, use the following command with the parameters that you require:

  Before IBM MQ 9.3.0:

  ```
  WmqDotnetXAMonitor.exe -m QueueManagerName -n ConnectionName -c ChannelName -i
  ```

  From IBM MQ 9.3.0:

  ```
  WmqDotnetXAMonitor.exe -m QueueManagerName -n ConnectionName -c ChannelName -i -k SSL Key
  Repository -s Cipher Spec
  ```

  The parameters that you can specify are as follows:

  – **-m *QueueManagerName***
  The queue manager name.

  Optional

  -n *ConnectionName*
  The connection name in host (port) format. *ConnectionName* can contain more than one connection name. Multiple connection names must be given in a comma separated list, for example `localhost (1414), localhost (1415), localhost (1416)`. The WMQDotnetXAMonitor application runs the recovery for each of the connection names specified in the comma separated list.

-c *ChannelName*
The channel name.

-i
Heuristic branch completion.

Optional

-k *SSL Key Repository*
The name of the SSL key repository. The supported values are:

- *SYSTEM (this is the default value)
- *USER

Optional

-s *Cipher Spec*
The CipherSpec that you set must be one of the CipherSpecs for the supported version and it can preferably be the same as the one specified in the Windows Group Policy. For more information, see "CipherSpec support for the managed .NET client" on page 578.

Mandatory for establishing a secure connection to the queue manager.

-dn *SSLPeer Name*
The SSL peer name used to check the Distinguished Name (DN) of the certificate from the peer queue manager.

Optional

-cl *Certificate Label*
The label name that identifies the certificate.

Optional

-sn *OutboundSNI*
Whether the Server Name Indication (SNI) should be set to the target IBM MQ channel name to the remote system when initiating a TLS connection, or to the hostname. The supported values for this option are:

- CHANNEL (this is the default value)
- HOSTNAME
- *

If no value is set then the default value, that is `CHANNEL`, is used.

Optional

-cr *Certificate Revocation Check*
Whether certification revocation checking is to be done. The supported values for this option are:

- true
- false (this is the default value)

Optional

-kr *KeyResetCount*
The total number of unencrypted bytes that are sent and received on the channel before the secret key used for encryption is renegotiated.

The default value of 0 indicates that secret keys are never renegotiated

Optional

The WMQDotnetXAMonitor application performs the following actions:

1. Checks the queue depth of SYSTEM.DOTNET.XARECOVERY.QUEUE at an interval of 100 seconds.
2. If the queue depth is greater than zero, browses the queue for messages and checks if the messages satisfy the incomplete transaction criteria.

3. If a message satisfies the incomplete transaction criteria, pulls it out, and retrieves the transaction recovery information.
4. Determines if the recovery information relates to the local Microsoft Distributed Transaction coordinator (MS DTC). If this is the case, then WMQDotnetXAMonitor proceeds to recover the transaction, otherwise it goes back to browse the next message.
5. Makes calls to the queue manager to recover the incomplete transaction.

WmqDotNETXAMonitor application configuration file settings

You can provide input to the IBM MQ .NET XA Monitor application, WmqDotNETXAMonitor, by using an application configuration file. A sample application configuration file is shipped with IBM MQ .NET. You can modify this sample file according to your requirements.

Input values provided through the application configuration file take the highest precedence. If you provide input values both at the command line as described in "Using the WMQDotnetXAMonitor application" on page 556 and in the application configuration file, then the values from the application configuration file take precedence.

Sample application configuration file for before IBM MQ 9.3.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<configSections>
<sectionGroup name="IBM.WMQ">
<section name="dnetxa" type="System.Configuration.NameValueFileSectionHandler" />
</sectionGroup>
</configSections>
<IBM.WMQ>
<dnetxa>
<add key="ConnectionName" value=""/>
<add key="ChannelName" value="" />
<add key="QueueManagerName" value="" />
<add key="UserId" value="" />
<add key="SecurityExit" value="" />
<add key="SecurityExitUserData" value = "">
</dnetxa>
</dnetxa>
</configuration>
```

Sample application configuration file from IBM MQ 9.3.0.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<configSections>
<sectionGroup name="IBM.WMQ">
<section name="dnetxa" type="System.Configuration.NameValueFileSectionHandler" />
</sectionGroup>
</configSections>
<IBM.WMQ>
<dnetxa>
<add key="ConnectionName" value=""/>
<add key="ChannelName" value="" />
<add key="QueueManagerName" value="" />
<add key="UserId" value="" />
<add key="SecurityExit" value="" />
<add key="SecurityExitUserData" value = "">
<add key="SSLKeyRepository" value="" />
<add key="SSLCipherSpec" value="" />
<add key="SSLPeerName" value="" />
<add key="SSLKeyResetCount" value="" />
<add key="SSLCertRevocationCheck" value="" />
<add key="CertificateLabel" value="" />
<add key="OutboundSNI" value="" />
</dnetxa>
</dnetxa>
</configuration>
```

WmqDotNetXAMonitor Application log
Monitor Application creates a log file in the application directory for logging the Monitor's progress and transaction recovery status. Logging starts with the connection name and the channel details to show current queue manager for which the recovery is running.

Once the recovery starts, MessageId of the transaction recovery message, TransactionId of the incomplete transaction and actual outcome of the transaction as per Transaction Manager Coordination will be logged.

Sample log file:

```
Time|ProcessId|ThreadId|WMQ .NET XA Recovery Monitor, Running now for
ConnectionName:xxxx, Time|ProcessId|ThreadId|Channel=xxxx
Time|ProcessId|ThreadId|Current QueueDepth = n
Time|ProcessId|ThreadId|Current MessageId = xxxx
Time|ProcessId|ThreadId|Current Incomplete Transaction being recovered = xxxxx
Time|ProcessId|ThreadId|Actual Outcome of the transaction(as per DTC)= Commit/Roll back
Time|ProcessId|ThreadId|Recovery Completed for TransactionId= xxxxx
Time|ProcessId|ThreadId|Current QueueDepth = n
Time|ProcessId|ThreadId|Current MessageId = xxxx
Time|ProcessId|ThreadId|Current Incomplete Transaction being recovered = xxxxx
Time|ProcessId|ThreadId|Actual Outcome of the transaction(as per DTC)= Commit/Roll back
Time|ProcessId|ThreadId| Recovery Completed for TransactionId= xxxxx
```

Writing and deploying IBM MQ .NET programs

To use IBM MQ classes for .NET to access IBM MQ queues, you write programs in any language supported by .NET containing calls that put messages onto, and get messages from, IBM MQ queues.

Before you begin

Deprecated V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, in IBM MQ classes for .NET, the methods WriteObject(), ReadObject(), CreateObjectMessage(), and the classes ObjectMessage and XmsObjectMessageImpl used for serialization and deserialization of data are deprecated.

Removed V 9.4.0 V 9.4.0 The IBM MQ .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

About this task

The IBM MQ documentation contains information only on the C#, C++ and Visual Basic languages.

The topics in this section provide information to assist with writing applications to interact with IBM MQ systems. For details of individual classes, see The IBM MQ .NET classes and interfaces.

Connection differences

The way that you program for IBM MQ.NET has some dependencies on the connection modes that you want to use.

When IBM MQ classes for .NET are used as a managed client, there are a number of differences from a standard IBM MQ MQI client, as some features are not available to a managed client.

IBM MQ.NET determines which connection type to use from the settings that you specify for the connection name, channel name, the customization value NMQ_MQ_LIB and the property MQC.TRANSPORT_PROPERTY.

Managed client connections

When IBM MQ classes for .NET are used as a managed client, there are a number of differences from a standard IBM MQ MQI client.

The following features are not available to a managed client:

- Channel compression

- Channel exit chaining

If you try to use these features with a managed client, it will return an MQException. If the error is detected at the client end of a connection, it will use reason code MQRC_ENVIRONMENT_ERROR. If it is detected at the server end, the reason code returned by the server will be used.

Channel exits written for an unmanaged client do not work. You must write new exits specifically for the managed client. Check that there are no invalid channel exits specified in your client channel definition table (CCDT).

The name of a managed channel exit can be up to 999 characters long. However, if you use the CCDT to specify the channel exit name, it is limited to 128 characters.

Communication is supported only over TCP/IP.

When you stop a queue manager using the **endmqm** command, a server-connection channel to a .NET managed client can take longer to close than server-connection channels to other clients.

If you are have set *NMQ_MQ_LIB* to `managed` in order to use managed IBM MQ problem diagnostics, none of the parameters -i, -p, -s, -b, or -c of the **strmqtrc** command is supported.

A managed .NET application using XA transactions will not work with a z/OS queue manager.
A managed .NET client attempting to connect to a z/OS queue manager fails with an error, MQRC_UOW_ENLISTMENT_ERROR (mqrc=2354), on MQOPEN call. However, a managed .NET application using XA transactions will work with distributed queue manager.

Defining which connection type to use

The connection type is determined by the setting of the connection name, channel name, the customization value NMQ_MQ_LIB and the property MQC.TRANSPORT_PROPERTY.

You can specify the connection name as follows:

- Explicitly on an MQQueueManager constructor:

```
public MQQueueManager(String queueManagerName, MQLONG Options, string Channel,
string ConnName)
```

```
public MQQueueManager(String queueManagerName, string Channel, string ConnName)
```

- By setting the properties MQC.HOST_NAME_PROPERTY and, optionally, MQC.PORT_PROPERTY in a hashtable entry on an MQQueueManager constructor:

```
public MQQueueManager(String queueManagerName, Hashtable properties)
```

- As explicit MQEnvironment values

```
MQEnvironment.Hostname
```

`MQEnvironment.Port` (optional).

- By setting the properties MQC.HOST_NAME_PROPERTY and, optionally, MQC.PORT_PROPERTY in the MQEnvironment.properties hashtable.

You can specify the channel name as follows:

- Explicitly on an MQQueueManager constructor:

```
public MQQueueManager(String queueManagerName, MQLONG Options, string Channel,
string ConnName)
```

```
public MQQueueManager(String queueManagerName, string Channel, string ConnName)
```

- By setting the property MQC.CHANNEL_PROPERTY in a hashtable entry on an MQQueueManager constructor:

```
public MQQueueManager(String queueManagerName, Hashtable properties)
```

- As an explicit MQEnvironment value

```
MQEnvironment.Channel
```

- By setting the property MQC.CHANNEL_PROPERTY in the MQEnvironment.properties hashtable.

You can specify the transport property as follows:

- By setting the property MQC.TRANSPORT_PROPERTY in a hashtable entry on an MQQueueManager constructor:

```
public MQQueueManager(String queueManagerName, Hashtable properties)
```

- By setting the property MQC.TRANSPORT_PROPERTY in the MQEnvironment.properties hashtable.

Select the connection type you require by using one of the following values:

MQC.TRANSPORT_MQSERIES_BINDINGS - connect as server
MQC.TRANSPORT_MQSERIES_CLIENT - connect as non-XA client
MQC.TRANSPORT_MQSERIES_XACLIENT - connect as XA client
MQC.TRANSPORT_MQSERIES_MANAGED - connect as non-XA managed client

You can set the customization value NMQ_MQ_LIB to explicitly choose the connection type as shown in the following table.

NMQ_MQ_LIB value	**Connection type**
mqic.dll	Connect as a non-XA client
mqicxa.dll	Connect as an XA client
mqm.dll	Connect as a server or as a non-XA client
managed	Connect as a non-XA managed client

Note: Values of mqic32.dll and mqic32xa.dll are accepted as synonyms of mqic.dll and mqicxa.dll for compatibility with earlier releases. However, mqm.dll and mqm.pdb are only part of the client package from IBM WebSphere MQ 7.1 onwards.

If you choose a connection type which is unavailable in your environment, for example you specify mqic32xa.dll and do not have XA support, IBM MQ.NET throws an exception.

Setting NMQ_MQ_LIB to "managed" causes the client to use managed IBM MQ problem diagnostic tests, .NET data conversion, and other managed low-level IBM MQ functions.

All other values for NMQ_MQ_LIB cause the .NET process to use unmanaged IBM MQ problem diagnostic tests and data conversion, and other unmanaged low-level IBM MQ functions (assuming an IBM MQ MQI client or server is installed on the system).

IBM MQ.NET chooses the connection type as follows:

1. If MQC.TRANSPORT_PROPERTY is specified, it connects according to the value of MQC.TRANSPORT_PROPERTY.

   Note, however, that setting MQC.TRANSPORT_PROPERTY to MQC.TRANSPORT_MQSERIES_MANAGED does not guarantee that the client process runs managed. Even with this setting, the client is not managed in the following cases:

   - If another thread in the process has connected with MQC.TRANSPORT_PROPERTY set to something other than MQC.TRANSPORT_MQSERIES_MANAGED.

- If NMQ_MQ_LIB is not set to "managed", problem diagnostic tests, data conversion, and other low-level functions are not fully managed (assuming an IBM MQ MQI client or server is installed on the system).

2. If a connection name has been specified without a channel name, or a channel name has been specified without a connection name, it throws an error.
3. If both a connection name and a channel name have been specified:
   - If NMQ_MQ_LIB is set to mqic32xa.dll, it connects as an XA client.
   - If NMQ_MQ_LIB is set to managed, it connects as a managed client.
   - Otherwise it connects as a non-XA client.
4. If NMQ_MQ_LIB is specified, it connects according to the value of NMQ_MQ_LIB.
5. If an IBM MQ server is installed, it connects as a server.
6. If an IBM MQ MQI client is installed, it connects as a non-XA client.
7. Otherwise, it connects as a managed client.

Windows Using the IBM MQ .NET project template

The IBM MQ .NET client offers you the ability to use a project template to assist you in developing your .NET Core applications.

Before you begin

You must have Microsoft Visual Studio 2017, or later, and .NET Core 2.1 on your system.

You must copy the .NET template from the

&MQ_INSTALL_ROOT&\tools\dotnet\samples\cs\core\base\ProjectTemplates\IBMMQ.NETClientApp.zip

directory to the

&USER_HOME_DIRECTORY&\Documents\&Visual_Studio_Version&\Templates\ProjectTemplates

directory, where:

- *&MQ_INSTALL_ROOT* is the root directory of your installation
- *&USER_HOME_DIRECTORY* is your home directory.

You must stop and restart Microsoft Visual Studio to pick up the template.

About this task

The .NET project template includes some common code that you can use to help develop your applications. With the in-built code, you can connect to the IBM MQ queue manager, and perform a put or a get operation by simply modifying the properties in the in-built code.

Procedure

1. Open Microsoft Visual Studio.
2. Click on **File**, followed by **New** and then **Project**.
3. In the *Create a new project window*, select `IBM MQ .NET Client App (.NET Core)` and click **Next**.
4. In the *Configure your new project* window, change the *Project name* of your project if you want to, and click **Create** to create the .NET project.

   `MQDotnetApp.cs` is the file that is created along with the project file. This file contains the code which connects to the queue manager, and performs a put and get operation.

The connection properties are set to default values:

- MQC.CONNECTION_NAME_PROPERTY is set to *localhost(1414)*
- MQC.CHANNEL_PROPERTY is set to *DOTNET.SVRCONN*

The queue is set to *Q1*, and you can modify these properties accordingly.

5. Compile and run the application.

Related concepts

IBM MQ components and features

.NET application runtime - Windows only

Configuration files for IBM MQ classes for .NET

A .NET client application can use an IBM MQ MQI client configuration file and, if you are using the managed connection type, a .NET application configuration file. Settings in the application configuration file have priority.

Client configuration file

An IBM MQ classes for .NET client application can use a client configuration file in the same way as any other IBM MQ MQI client. This file is typically called `mqclient.ini`, but you can specify a different file name. For more information about the client configuration file, see IBM MQ MQI client configuration file, `mqclient.ini`.

Only the following attributes in an IBM MQ MQI client configuration file are relevant to IBM MQ classes for .NET. If you specify other attributes, it has no effect.

Table 77. Client configuration file attributes that are relevant to IBM MQ classes for .NET

Stanza	Attribute
CHANNELS	CCSID
CHANNELS	ChannelDefinitionDirectory
CHANNELS	ChannelDefinitionFile
CHANNELS	ReconDelay
CHANNELS	DefRecon
CHANNELS	MQReconnectTimeout
CHANNELS	ServerConnectionParms
CHANNELS	Put1DefaultAlwaysSync
CHANNELS	PasswordProtection
ClientExitPath	ExitsDefaultPath
ClientExitPath	ExitsDefaultPath64
MessageBuffer	MaximumSize
MessageBuffer	PurgeTime
MessageBuffer	UpdatePercentage
Security	MQIInitialKeyFile
SSL	SSLKeyRepository
SSL	SSLKeyRepositoryPassword
TCP	ClntRcvBufSize

Table 77. Client configuration file attributes that are relevant to IBM MQ classes for .NET (continued)	
Stanza	**Attribute**
TCP	ClntSndBufSize
TCP	IPAddressVersion
TCP	KeepAlive

You can override any of these attributes using the appropriate environment variable.

Application configuration file

If you are running with the managed connection type, you can also override the IBM MQ client configuration file and the equivalent environment variables using the .NET application configuration file.

The .NET application configuration file settings are only acted upon when running with the managed connection type, and are ignored for other connection types.

The .NET application configuration file and its format are defined by Microsoft for general use within the .NET framework, but the particular section names, keys and values mentioned in this documentation are specific to IBM MQ.

The format of the .NET application configuration file is a number of *sections*. Each section contains one or more *keys*, and each key has an associated *value*. The following example shows the sections, keys, and values used in a .NET application configuration file to control the TCP/IP KeepAlive property:

```
<configuration>
  <configSections>
    <section name="TCP" type="System.Configuration.NameValueSectionHandler"/>
  </configSections>
  <TCP>
    <add key="KeepAlive" value="true"></add>
  </TCP>
<configuration>
```

The keywords used in the .NET application configuration file section names and keys exactly match the keywords for the stanzas and attributes defined in the client configuration file.

The section `<configSections>` must be the first child element of the `<configuration>` element.

See your Microsoft documentation for further information.

Example C# code fragment for use with .NET

A C# code fragment demonstrating that an application connects to a queue manager, puts a message on to a queue and receives a reply.

The following C# code fragment demonstrates an application that performs three actions:

1. Connect to a queue manager
2. Put a message onto SYSTEM.DEFAULT.LOCAL.QUEUE
3. Get the message back

It also shows how to change the connection type.

```
// ============================================================================
// Licensed Materials - Property of IBM
// 5724-H72
// (c) Copyright IBM Corp. 2003, 2024
// ============================================================================
using System;
using System.Collections;

using IBM.WMQ;
```

```
class MQSample
{
  // The type of connection to use, this can be:-
  // MQC.TRANSPORT_MQSERIES_BINDINGS for a server connection.
  // MQC.TRANSPORT_MQSERIES_CLIENT for a non-XA client connection
  // MQC.TRANSPORT_MQSERIES_XACLIENT for an XA client connection
  // MQC.TRANSPORT_MQSERIES_MANAGED for a managed client connection
  const String connectionType = MQC.TRANSPORT_MQSERIES_CLIENT;

  // Define the name of the queue manager to use (applies to all connections)
  const String qManager = "your_Q_manager";

  // Define the name of your host connection (applies to client connections only)
  const String hostName = "your_hostname";

  // Define the name of the channel to use (applies to client connections only)
  const String channel = "your_channelname";



  /// <summary>
  /// Initialise the connection properties for the connection type requested
  /// </summary>
  /// <param name="connectionType">One of the MQC.TRANSPORT_MQSERIES_ values</param>
  static Hashtable init(String connectionType)
  {
    Hashtable connectionProperties = new Hashtable();

    // Add the connection type
    connectionProperties.Add(MQC.TRANSPORT_PROPERTY, connectionType);

    // Set up the rest of the connection properties, based on the
    // connection type requested
    switch(connectionType)
    {
      case MQC.TRANSPORT_MQSERIES_BINDINGS:
        break;
      case MQC.TRANSPORT_MQSERIES_CLIENT:
      case MQC.TRANSPORT_MQSERIES_XACLIENT:
      case MQC.TRANSPORT_MQSERIES_MANAGED:
        connectionProperties.Add(MQC.HOST_NAME_PROPERTY, hostName);
        connectionProperties.Add(MQC.CHANNEL_PROPERTY, channel);
        break;
    }

    return connectionProperties;
  }
  /// <summary>
  /// The main entry point for the application.
  /// </summary>
  [STAThread]
  static int Main(string[] args)
  {
    try
    {
      Hashtable connectionProperties = init(connectionType);

      // Create a connection to the queue manager using the connection
      // properties just defined
      MQQueueManager qMgr = new MQQueueManager(qManager, connectionProperties);

      // Set up the options on the queue we want to open
      int openOptions = MQC.MQOO_INPUT_AS_Q_DEF | MQC.MQOO_OUTPUT;

      // Now specify the queue that we want to open,and the open options
      MQQueue system_default_local_queue =
        qMgr.AccessQueue("SYSTEM.DEFAULT.LOCAL.QUEUE", openOptions);

      // Define an IBM MQ message, writing some text in UTF format
      MQMessage hello_world = new MQMessage();
      hello_world.WriteUTF("Hello World!");

      // Specify the message options
      MQPutMessageOptions pmo = new MQPutMessageOptions(); // accept the defaults,
                                                           // same as MQPMO_DEFAULT

      // Put the message on the queue
      system_default_local_queue.Put(hello_world, pmo);
```

```
      // Get the message back again

      // First define an IBM MQ message buffer to receive the message
      MQMessage retrievedMessage =new MQMessage();
      retrievedMessage.MessageId =hello_world.MessageId;

      // Set the get message options
      MQGetMessageOptions gmo =new MQGetMessageOptions(); //accept the defaults
                                                          //same as MQGMO_DEFAULT

      // Get the message off the queue
      system_default_local_queue.Get(retrievedMessage,gmo);

      // Prove we have the message by displaying the UTF message text
      String msgText = retrievedMessage.ReadUTF();
      Console.WriteLine("The message is: {0}", msgText);

      // Close the queue
      system_default_local_queue.Close();

      // Disconnect from the queue manager
      qMgr.Disconnect();
    }

    //If an error has occurred,try to identify what went wrong.

    //Was it an IBM MQ error?
    catch (MQException ex)
    {
      Console.WriteLine("An IBM MQ error occurred: {0}", ex.ToString());
    }

    catch (System.Exception ex)
    {
      Console.WriteLine("A System error occurred: {0}", ex.ToString());
    }

    return 0;
  }//end of start
}//end of sample
```

Setting up the IBM MQ environment

Before you use the client connection to connect to a queue manager, you must set up the IBM MQ environment.

Note: This step is not necessary when using IBM MQ classes for .NET in server bindings mode.

The .NET programming interface allows you to use the NMQ_MQ_LIB customization value but also includes a class MQEnvironment. This class allows you to specify details that are to be used during the connection attempt, such as those in the following list:

- Channel name
- Host name
- Port number
- Channel exits
- SSL parameters
- User ID and password

For full information about the MQEnvironment class, see MQEnvironment.NET class

To specify the channel name and host name, use the following code:

```
MQEnvironment.Hostname = "host.domain.com";
MQEnvironment.Channel  = "client.channel";
```

By default, the clients attempt to connect to a IBM MQ listener at port 1414. To specify a different port, use the code:

```
MQEnvironment.Port = nnnn;
```

Connecting to and disconnecting from a queue manager

When you have configured the IBM MQ environment, you are ready to connect to a queue manager.

To connect to a queue manager, create a new instance of the MQQueueManager class:

```
MQQueueManager queueManager = new MQQueueManager("qMgrName");
```

To disconnect from a queue manager, call the `Disconnect` method on the queue manager:

```
queueManager.Disconnect();
```

You must have inquire (`inq`) authority on the queue manager when attempting to connect to the queue manager. Without inquire authority, the connection attempt fails.

If you call the `Disconnect` method, all open queues and processes that you have accessed through that queue manager are closed. However, it is good programming practice to close these resources explicitly when you finish using them. To close the resources, use the `Close` method on the object associated with each resource.

The `Commit` and `Backout` methods on a queue manager replace the MQCMIT and MQBACK calls that are used with the procedural interface.

Accessing queues and topics

You can access queues and topics using methods of MQQueueManager or appropriate constructors.

To access queues, use the methods of the MQQueueManager class. The MQOD (object descriptor structure) is collapsed into the parameters of these methods. For example, to open a queue on a queue manager represented by an MQQueueManager object called queueManager, use the following code:

```
MQQueue queue = queueManager.AccessQueue("qName",
                                         MQC.MQOO_OUTPUT,
                                         "qMgrName",
                                         "dynamicQName",
                                         "altUserId");
```

The *options* parameter is the same as the Options parameter in the MQOPEN call.

The AccessQueue method returns a new object of class MQQueue.

When you have finished using the queue, use the Close() method to close it, as in the following example:

```
queue.Close();
```

With IBM MQ .NET, you can also create a queue by using the MQQueue constructor. The parameters are exactly the same as for the accessQueue method, with the addition of a queue manager parameter specifying the instantiated MQQueueManager object to use. For example:

```
MQQueue queue = new MQQueue(queueManager,
                            "qName",
                            MQC.MQOO_OUTPUT,
                            "qMgrName",
                            "dynamicQName",
                            "altUserId");
```

Constructing a queue object in this way enables you to write your own subclasses of MQQueue.

Similarly, you can also access topics using the methods of the MQQueueManager class. Use an AccessTopic() method to open a topic. This returns a new object of class MQTopic. When you have finished using the topic, use the Close() method of the MQTopic to close it.

You can also create a topic by using an MQTopic constructor. There are a number of constructors for topics; for more information see MQTopic.NET class.

Handling messages

Messages are handled using the methods of the queue or topic classes. To build a new message, create a new MQMessageobject.

Put messages onto queues or topics using the Put() method of the MQQueue or MQTopic class. Get messages from queues or topics using the Get() method of the MQQueue or MQTopic class. Unlike the procedural interface, where MQPUT and MQGET put and get arrays of bytes, the IBM MQ classes for .NET put and get instances of the MQMessage class. The MQMessage class encapsulates the data buffer that contains the actual message data, together with all the MQMD (message descriptor) parameters that describe that message.

To build a new message, create a new instance of the MQMessage class and use the WriteXXX methods to put data into the message buffer.

When the new message instance is created, all the MQMD parameters are automatically set to their default values, as defined in Initial values and language declarations for MQMD. The Put() method of MQQueue also takes an instance of the MQPutMessageOptions class as a parameter. This class represents the MQPMO structure. The following example creates a message and puts it onto a queue:

```
// Build a new message containing my age followed by my name
MQMessage myMessage = new MQMessage();
myMessage.WriteInt(25);

String name = "Charlie Jordan";
myMessage.WriteUTF(name);

// Use the default put message options...
MQPutMessageOptions pmo = new MQPutMessageOptions();

// put the message
!queue.Put(myMessage,pmo);
```

The Get() method of MQQueue returns a new instance of MQMessage, which represents the message just taken from the queue. It also takes an instance of the MQGetMessageOptions class as a parameter. This class represents the MQGMO structure.

You do not need to specify a maximum message size, because the Get() method automatically adjusts the size of its internal buffer to fit the incoming message. Use the ReadXXX methods of the MQMessage class to access the data in the returned message.

The following example shows how to get a message from a queue:

```
// Get a message from the queue
MQMessage theMessage    = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();
queue.Get(theMessage,gmo);  // has default values

// Extract the message data
int age = theMessage.ReadInt();
String name1 = theMessage.ReadUTF();
```

You can alter the number format that the read and write methods use by setting the *encoding* member variable.

You can alter the character set to use for reading and writing strings by setting the *characterSet* member variable.

See MQMessage.NET class for more details.

Note: The WriteUTF() method of MQMessage automatically encodes the length of the string as well as the Unicode bytes it contains. When your message will be read by another .NET program (using ReadUTF()), this is the simplest way to send string information.

Handling message properties

Message properties allow you to select messages, or to retrieve information about a message without accessing its headers. The MQMessage class contains methods to get and set properties.

You can 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 IBM MQ and JMS applications. For more information about message properties in IBM MQ, see Message properties.

The MQMessage class provides a number of methods to get and set properties, according to the data type of the property. The get methods have names of the format Get*Property, and the set methods have names of the format Set*Property, where the asterisk (*) represents one of the following strings:

- Boolean
- Byte
- Bytes
- Double
- Float
- Int
- Int2
- Int4
- Int8
- Long
- Object
- Short
- String

For example, to get the IBM MQ property myproperty (a character string), use the call `message.GetStringProperty('myproperty')`. You can optionally pass a property descriptor, which IBM MQ will complete.

Handling errors

Handle errors arising from IBM MQ classes for .NET using `try` and `catch` blocks.

Methods in the .NET interface do not return a completion code and reason code. Instead, they throw an exception whenever the completion code and reason code resulting from an IBM MQ call are not both zero. This simplifies the program logic so that you do not have to check the return codes after each call to IBM MQ. You can decide at which points in your program you want to deal with the possibility of failure. At these points, you can surround your code with `try` and `catch` blocks, as in the following example:

```
try
{
  myQueue.Put(messageA,PutMessageOptionsA);
  myQueue.Put(messageB,PutMessageOptionsB);
}
catch (MQException ex)
{
  // This block of code is only executed if one of
  // the two put methods gave rise to a non-zero
  // completion code or reason code.
  Console.WriteLine("An error occurred during the put operation:" +
                    "CC = " + ex.CompletionCode +
                    "RC = " + ex.ReasonCode);
```

```
  Console.WriteLine("Cause exception:" + ex );
}
```

Getting and setting attribute values

The classes MQManagedObject, MQQueue, and MQQueueManager contain methods that allow you to get and set their attribute values. Note that for MQQueue, the methods work only if you specify the appropriate inquire and set flags when you open the queue.

For common attributes, the MQQueueManager and MQQueue classes inherit from a class called MQManagedObject. This class defines the Inquire() and Set() interfaces.

When you create a new queue manager object by using the *new* operator, it is automatically opened for inquire. When you use the AccessQueue() method to access a queue object, that object is not automatically opened for either inquire or set operations, this could cause problems with some types of remote queues. To use the Inquire and Set methods and to set properties on a queue, you must specify the appropriate inquire and set flags in the openOptions parameter of the AccessQueue() method.

The inquire and set methods take three parameters:

- selectors array
- intAttrs array
- charAttrs array

You do not need the SelectorCount, IntAttrCount, and CharAttrLength parameters that are found in MQINQ, because the length of an array is always known. The following example shows how to make an inquiry on a queue:

```
//inquire on a queue
int [ ] selectors = new int [2] ;
int [ ] intAttrs = new int [1] ;
byte [ ] charAttrs = new byte [MQC.MQ_Q_DESC_LENGTH];
selectors [0] = MQC.MQIA_DEF_PRIORITY;
selectors [1] = MQC.MQCA_Q_DESC;
queue.Inquire(selectors,intAttrs,charAttrs);
ASCIIEncoding enc = new ASCIIEncoding();
String s1 = "";
s1 = enc.GetString(charAttrs);
```

All attributes of these objects can be inquired on. A subset of attributes is exposed as the properties of an object. For a list of object attributes, see Attributes of objects. For object properties, see the appropriate class description.

Multithreaded programs

The .NET runtime environment is inherently multithreaded. IBM MQ classes for .NET allows a queue manager object to be shared across multiple threads but ensures that all access to the target queue manager is synchronized.

Consider a simple program that connects to a queue manager and opens a queue at startup. The program displays a single button on the screen. When a user clicks that button, the program fetches a message from the queue. In this situation, the application initialization occurs in one thread, and the code that executes in response to the button press executes in a separate thread (the user interface thread).

The implementation of IBM MQ .NET ensures that, for a particular connection (MQQueueManager object instance), all access to the target IBM MQ queue manager is synchronized. The default behavior is that a thread that wants to issue a call to a queue manager is blocked until all other calls in progress for that connection are complete. If you require simultaneous access to the same queue manager from multiple threads within your program, create a new MQQueueManager object for each thread that requires concurrent access. (This is equivalent to issuing a separate MQCONN call for each thread.)

If the default connection options are overridden by MQC.MQCNO_HANDLE_SHARE_NONE or MQC.MQCNO_SHARE_NO_BLOCK then the queue manager is no longer synchronized.

Using a client channel definition table with .NET

You can use a client channel definition table (CCDT) with the IBM MQ classes for .NET. You specify the location of the CCDT in different ways, depending on whether you are using a managed or unmanaged connection.

Non-XA or XA unmanaged client connection type

With an unmanaged connection type, you can specify the location of the CCDT in two ways:

- Using the environment variables MQCHLLIB to specify the directory where the table is located, and MQCHLTAB to specify the file name of the table.
- Using the client configuration file. In the CHANNELS stanza, use the attributes **ChannelDefinitionDirectory** to specify the directory where the table is located, and **ChannelDefinitionFile** to specify the file name.

If the location is specified both in the client configuration file and by using environment variables, the environment variables take priority. You can use this feature to specify a standard location in the client configuration file and override it using environment variables when necessary.

Managed client connection type

With a managed connection type, you can specify the location of the CCDT in three ways:

- Using the .NET application configuration file. In the CHANNELS section, use the keys **ChannelDefinitionDirectory** to specify the directory where the table is located, and **ChannelDefinitionFile** to specify the file name.
- Using the environment variables MQCHLLIB to specify the directory where the table is located, and MQCHLTAB to specify the file name of the table.
- Using the client configuration file. In the CHANNELS stanza, use the attributes **ChannelDefinitionDirectory** to specify the directory where the table is located, and **ChannelDefinitionFile** to specify the file name.

If the location is specified in more than one of these ways, the environment variables take priority over the client configuration file, and the .NET Application Configuration File takes priority over both other methods. You can use this feature to specify a standard location in the client configuration file and override it using environment variables or the application configuration file when necessary.

From IBM MQ 9.3.0, the .NET client behaves in the same way as the C and Java clients and returns the MQRC_Q_MGR_NAME_ERROR when using a CCDT with queue manager grouping.

How a .NET application determines what channel definition to use

In the IBM MQ .NET client environment, the channel definition to be used can be specified in a number of different ways. Multiple specifications of the channel definition can exist. An application derives the channel definition from one or more sources.

If more than one channel definition exists, the one used is selected in the following priority order:

1. Properties specified on the MQQueueManager constructor, either explicitly or by including *MQC.CHANNEL_PROPERTY* in the properties hashtable
2. A property *MQC.CHANNEL_PROPERTY* in the MQEnvironment.properties hashtable
3. The property *Channel* in MQEnvironment
4. The .NET application configuration file, section name CHANNELS, key ServerConnectionParms (applies to managed connections only)
5. The *MQSERVER* environment variable
6. The client configuration file, stanza CHANNELS, Attribute ServerConnectionParms
7. The client channel definition table (CCDT). The location of the CCDT is specified in the .NET application configuration file (applies to managed connections only)

8. The client channel definition table (CCDT). The location of the CCDT is specified using the environment variables *MQCHLIB* and *MQCHLTAB*
9. The client channel definition table (CCDT). The location of the CCDT is specified using the client configuration file

For items 1-3, the channel definition is built up field by field from values provided by the application. These values can be provided using different interfaces and multiple values can exist for each one. Field values are added to the channel definition following the priority order given:

1. The value of *connName* on the MQQueueManager constructor
2. Values of properties from the MQQueueManager.properties hashtable
3. Values of properties from the MQEnvironment.properties hashtable
4. Values set as MQEnvironment fields (for example, MQEnvironment.Hostname, MQEnvironment.Port)

For items 4-6, the entire channel definition is supplied as the value. Unspecified fields on the channel definition take the system defaults. No values from other methods of defining channels and their fields are merged with these specifications.

For items 7-9, the entire channel definition is taken from the CCDT. Fields which were not specified explicitly when the channel was defined take the system defaults. No values from other methods of defining channels and their fields are merged with these specifications.

Using channel exits in IBM MQ .NET

If you use client bindings, you can use channel exits as for any other client connection. If you use managed bindings, you must write an exit program that implements an appropriate interface.

Client bindings

If you use client bindings, you can use channel exits as described in Channel exits. You cannot use channel exits written for managed bindings.

Managed bindings

If you use a managed connection, to implement an exit, you define a new .NET class that implements the appropriate interface. Three exit interfaces are defined in the IBM MQ package:

- MQSendExit
- MQReceiveExit
- MQSecurityExit

Note: User exits written using these interfaces are not supported as channel exits in the unmanaged environment.

The following sample defines a class that implements all three:

```
class MyMQExits : MQSendExit, MQReceiveExit, MQSecurityExit
{
   // This method comes from the send exit
  byte[] SendExit(MQChannelExit       channelExitParms,
                  MQChannelDefinition channelDefinition,
                  byte[]              dataBuffer
                  ref int             dataOffset
                  ref int             dataLength
                  ref int             dataMaxLength)
  {
     // complete the body of the send exit here
  }


  // This method comes from the receive exit
  byte[] ReceiveExit(MQChannelExit       channelExitParms,
                     MQChannelDefinition channelDefinition,
                     byte[]              dataBuffer
                     ref int             dataOffset
```

```
                      ref int             dataLength
                      ref int             dataMaxLength)
  {
     // complete the body of the receive exit here
  }


  // This method comes from the security exit
  byte[] SecurityExit(MQChannelExit       channelExitParms,
                      MQChannelDefinition channelDefParms,
                      byte[]              dataBuffer
                      ref int             dataOffset
                      ref int             dataLength
                      ref int             dataMaxLength)
  {
     // complete the body of the security exit here
  }

}
```

Each exit is passed an MQChannelExit and an MQChannelDefinition object instance. These objects represent the MQCXP and MQCD structures defined in the procedural interface.

The data to be sent by a send exit, and the data received in a security or receive exit is specified using the exit's parameters.

On entry, the data at offset *dataOffset* with length *dataLength* in the byte array *dataBuffer* is the data that is about to be sent by a send exit, and the data received in a security or receive exit. The parameter *dataMaxLength* gives the maximum length (from *dataOffset*) available to the exit in *dataBuffer*. Note: For a security exit, it is possible for the dataBuffer to be null, if this is the first time the exit is called or the partner end elected to send no data.

On return, the value of *dataOffset* and *dataLength* should be set to point to the offset and length within the returned byte array that the .NET classes should then use. For a send exit, this indicates the data that it should send, and for a security or receive exit, the data that should be interpreted. The exit should normally return a byte array; exceptions are a security exit which could elect to send no data, and any exit called with the INIT or TERM reasons. The simplest form of exit that can be written therefore is one which does nothing more than return dataBuffer:

The simplest possible exit body is:

```
{
  return dataBuffer;
}
```

MQChannelDefinition class

The userid and password that are specified with the managed .NET client application are set in the IBM MQ .NET MQChannelDefinition class that is passed to the client security exit. The security exit copies the userid and password into the MQCD.RemoteUserIdentifier and MQCD.RemotePassword fields (see “Writing a security exit” on page 938).

Specifying channel exits (managed client)

If you specify a channel name and connection name when creating your MQQueueManager object (either in the MQEnvironment or on the MQQueueManager constructor) you can specify channel exits in two ways.

In order of precedence, these are:

1. Passing hashtable properties MQC.SECURITY_EXIT_PROPERTY, MQC.SEND_EXIT_PROPERTY or MQC.RECEIVE_EXIT_PROPERTY on the MQQueueManager constructor.
2. Setting the MQEnvironment SecurityExit, SendExit or ReceiveExit properties.

If you do not specify a channel name and connection name, the channel exits to use come from the channel definition picked up from a client channel definition table (CCDT). It is not possible to override

the values stored in the channel definition. See Client channel definition table and “Using a client channel definition table with .NET” on page 571 for more information about channel definition tables.

In each case, the specification takes the form of a string with the following format:

```
Assembly_name(Class_name)
```

Class_name is the fully qualified name, including namespace specification, of a .NET class that implements the IBM.WMQ.MQSecurityExit, IBM.WMQ.MQSendExit or IBM.WMQ.MQReceiveExit interface (as appropriate). *Assembly_name* is the fully qualified location, including file extension, of the assembly that houses the class. The length of the string is limited to 999 characters if you use the properties of MQEnvironment or MQQueueManager. However, if the channel exit name is specified in the CCDT, it is limited to 128 characters. When necessary, the .NET client code loads and creates an instance of the specified class by parsing the string specification.

Specifying channel exit user data (managed client)

Channel exits can have user data associated with them. If you specify a channel name and connection name when creating your MQQueueManager object (either in the MQEnvironment or on the MQQueueManager constructor) you can specify the user data in two ways.

In order of precedence, these are:

1. Passing hashtable properties MQC.SECURITY_USERDATA_PROPERTY, MQC.SEND_USERDATA_PROPERTY or MQC.RECEIVE_USERDATA_PROPERTY on the MQQueueManager constructor.
2. Setting the MQEnvironment SecurityUserData, SendUserData or ReceiveUserData properties.

If you do not specify a channel name and connection name, the exit user data values to use come from the channel definition picked up from the client channel definition table (CCDT). It is not possible to override the values stored in the channel definition. See Client channel definition table and “Using a client channel definition table with .NET” on page 571 for more information about channel definition tables.

In each case, the specification is a string, limited to 32 characters.

Automatic client reconnection in .NET

You can make your client reconnect automatically to a queue manager during an unexpected connection break.

A client can unexpectedly become disconnected from a queue manager if, for example, the queue manager stops, or the network or server fails.

Without automatic client reconnection, an error is produced when the connection fails. You can use the error code to help you reestablish the connection.

A client that uses the automatic client reconnection facility is called a reconnectable client. To create a reconnectable client, specify certain options called reconnect options while connecting to the queue manager.

If the client application is an IBM MQ .NET client, it can opt to get automatic client reconnection by specifying an appropriate value for CONNECT_OPTIONS_PROPERTY when you use the `MQQueueManager` class to create a queue manager. See Reconnection options for details of CONNECT_OPTIONS_PROPERTY values.

You can select whether the client application always connects and reconnects to a queue manager of the same name, to the same queue manager, or to any set of queue managers that are defined with the same `QMNAME` in the client connection table (See Queue Manager Groups in CCDT for details).

Transport Layer Security (TLS) support for .NET

IBM MQ classes for .NET client applications support Transport Layer Security (TLS) encryption. The TLS protocol provides communications security over the internet, and allow client/server applications to communicate in a way that is confidential and reliable.

Related concepts

IBM MQ.NET managed client TLS support

Cryptographic security protocols: TLS

TLS support for the unmanaged .NET client

TLS support for the unmanaged .NET client is based on the C MQI and IBM Global Security Kit (GSKit). The C MQI handles the TLS operations and GSKit implements the TLS secure socket protocols.

Enabling TLS for the unmanaged .NET client

TLS is supported only for client connections. To enable TLS, you must specify the CipherSpec to use when communicating with the queue manager, and this must match the CipherSpec set on the target channel.

To enable TLS, specify the CipherSpec using the SSLCipherSpec static member variable of MQEnvironment. The following example attaches to a SVRCONN channel named SECURE.SVRCONN.CHANNEL, which has been set up to require TLS with a CipherSpec of TLS_RSA_WITH_AES_128_CBC_SHA:

```
MQEnvironment.Hostname         = "your_hostname";
MQEnvironment.Channel          = "SECURE.SVRCONN.CHANNEL";
MQEnvironment.SSLCipherSpec    = "TLS_RSA_WITH_AES_128_CBC_SHA256";
MQEnvironment.SSLKeyRepository = "C:\mqm\key.kdb";
MQQueueManager qmgr = new MQQueueManager("your_Q_manager");
```

See Specifying CipherSpecs for a list of CipherSpecs.

The SSLCipherSpec property can also be set using the MQC.SSL_CIPHER_SPEC_PROPERTY in the hash table of connection properties.

To successfully connect using TLS, the client keystore must be set up with Certificate Authority root certificates chain from which the certificate presented by the queue manager can be authenticated. Similarly, if SSLClientAuth on the SVRCONN channel has been set to MQSSL_CLIENT_AUTH_REQUIRED, the client keystore must contain an identifying personal certificate that is trusted by the queue manager.

Using the Distinguished Name of the queue manager

The queue manager identifies itself using a TLS certificate, which contains a *Distinguished Name* (DN).

An IBM MQ .NET client application can use this DN to ensure that it is communicating with the correct queue manager. A DN pattern is specified using the sslPeerName variable of MQEnvironment. For example, setting:

```
MQEnvironment.SSLPeerName = "CN=QMGR.*, OU=IBM, OU=WEBSPHERE";
```

allows the connection to succeed only if the queue manager presents a certificate with a Common Name beginning QMGR., and at least two Organizational Unit names, the first of which must be IBM and the second WEBSPHERE.

The SSLPeerName property can also be set using the MQC.SSL_PEER_NAME_PROPERTY in the hash table of connection properties. For more information about Distinguished Names and rules for setting peer names, refer to Securing IBM MQ.

If SSLPeerName is set, connections succeed only if it is set to a valid pattern and the queue manager presents a matching certificate.

Error handling when using TLS

The following reason codes can be issued by IBM MQ classes for .NET when connecting to a queue manager using TLS:

MQRC_SSL_NOT_ALLOWED
The SSLCipherSpec property was set, but bindings connect was used. Only client connect supports TLS.

MQRC_SSL_PEER_NAME_MISMATCH
The DN pattern specified in the SSLPeerName property did not match the DN presented by the queue manager.

MQRC_SSL_PEER_NAME_ERROR
The DN pattern specified in the SSLPeerName property was not valid.

MQRC_KEY_REPOSITORY_ERROR
The location of the key repository is either not specified, not valid, or cannot be accessed.

TLS support for the managed .NET client

The managed .NET client uses the Microsoft .NET Framework libraries to implement TLS secure socket protocols. The Microsoft System.Net.SecuritySslStream class operates as a stream over connected TCP sockets and sends and receives data over that socket connection.

The minimum required .NET Framework level is .NET Framework v3.5. The level of Cipher Algorithm support is based on the .NET Framework level that the application is using:

- For applications that are based on .NET Framework levels 3.5 and 4.0, the available secure socket protocols are SSL 3.0 and TSL 1.0.
- For applications that are based on .NET Framework level 4.5, the available secure socket protocols are SSL 3.0, TLS 1.1 and TLS 1.2.

You might need to move applications that expect higher TLS protocol support to a later version of the framework as defined for Microsoft Security support in the .NET Framework.

The main features of TLS support for the managed .NET client are as follows:

TLS protocol support
TLS support for the .NET managed client is defined through the .NET SSLStream class, and depends on the .NET Framework that the application is using. For more information see "TLS protocol support for the managed .NET client" on page 578.

CipherSpec support
The TLS settings for the .NET managed client are as for the Microsoft.NET TLS steams. For more information see "CipherSpec support for the managed .NET client" on page 578 and "CipherSpec mappings for the managed .NET client" on page 580.

Key repositories
The key repository on the client side is a Windows keystore. The server side repository is a Cryptographic Message Syntax (CMS) type of repository. For more information see "Key repositories for the managed .NET client" on page 581.

Certificates
You can use self-signed TLS certificates to implement mutual authentication between a client and a queue manager. For more information see "Using certificates for the managed .NET client" on page 582.

SSLPEERNAME
In .NET, applications can use the optional SSLPEERNAME attribute to specify a Distinguished Name (DN) pattern. For more information see "SSLPEERNAME" on page 582.

FIPS compliance
Enabling FIPS programmatically is not supported by the Microsoft.NET Security library. FIPS enablement is controlled by the Windows Group Policy setting.

NSA Suite B compliance
IBM MQ implements RFC 6460. The Microsoft.NET implementation for NSA suite B is 5430. This is supported from .NET Framework 3.5 onwards.

Secret key reset or renegotiation
Although the SSLStream class does not support secret key resetting or renegotiation, for consistency with other IBM MQ clients, the .NET managed client allows applications to set `SSLKeyResetCount`. For more information see "Secret key reset or renegotiation for the managed .NET client" on page 583.

Revocation check
The SSLStream class supports certificate revocation checking, which is automatically done by the certificate chaining engine. For more information see "Revocation check" on page 583.

IBM MQ security exit support
The `SSLStream` class provides limited support for IBM MQ security exits. Querying local and remote certificates to get `SSLPeerNamePtr`(Subject DN) and `SSLRemCertIssNamePtr` (Issuer DN) is possible since this is supported in Microsoft.NET. However, there is no support for getting attributes like DNQ, UNSTRUCTUREDNAME and UNSTRUCTUREDADDRESS, so these values cannot be retrieved using the exits.

Cryptographic hardware support
Cryptographic hardware is not supported for the managed .NET client.

Support for TLS1.3 on managed IBM MQ .NET and XMS .NET clients

V 9.4.0

From IBM MQ 9.4.0, IBM MQ .NET and XMS .NET clients support TLS1.3 provided that the operating system supports TLS1.3.

The managed .NET client uses the Microsoft .NET Framework libraries to implement TLS secure socket protocols. The Microsoft `System.Net.SecuritySslStream` class operates as a stream over connected TCP sockets and sends and receives data over that socket connection.

On Windows, .NET uses SCHANNEL, and on Linux .NET uses OpenSSL for SSL Communication.

Windows For IBM MQ .NET client applications running on Windows

Microsoft had announced that Windows 11 and Windows Server 2022 support TSL1.3 ciphers by default.

TLS_AES_128_GCM_SHA256 and TLS_AES_256_GCM_SHA384 cipher suites are enabled by default on both versions of Windows.

Attention:

- TLS_CHACHA20_POLY1305_SHA256 Cipher Suite is not enabled by default but is supported.
- For an IBM MQ .NET client with TLS1.3 enabled, to connect to a queue manager successfully, IBM Global Security Kit (GSKit) 8.0.55.29 is the minimum version that is the required at the queue manager side.

Linux For IBM MQ .NET client applications running on Linux

As .NET uses OpenSSL on Linux for SSL Communication, to use TLS1.3, OpenSSL v1.1.1 is the minimum requirement.

Additionally, as .NET uses OpenSSL on Linux, all the ciphers supported by OpenSSL should work for .NET as well.

OpenSSL supports the following CipherSpecs for TLS1.3:

- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_AES_128_GCM_SHA256
- TLS_AES_128_CCM_8_SHA256
- TLS_AES_128_CCM_SHA256

Related concepts

“CipherSpec mappings for the managed .NET client” on page 580
The IBM MQ.NET interface maintains an IBM MQ to Microsoft.NET mapping table that is used to determine the version of the TLS protocol that the managed client needs to use to establish a secure connection with a queue manager.

TLS protocol support for the managed .NET client
IBM MQ.NET TLS support is based on the .NET SSLStream class.

Note: TLS protocol support for the managed .NET client depends on the .NET Framework level that the application is using. For more information, see “TLS support for the managed .NET client” on page 576.

For the Microsoft.NET SSLStream class to initialize TLS and perform a hand-shake with the queue manager, one of the required parameters that you must set is **SSLProtocol**, where you must specify the TLS version number, which must be one of the following values:

- SSL3.0
- TLS1.0
- TLS1.2

The value of this parameter is tightly coupled with the Protocol family to which the preferred CipherSpec belongs. When SSLStream starts an TLS handshake with the server (queue manager), it uses the TLS version specified in **SSLProtocol** to identify list of CipherSpecs to be used for negotiation.

IBM MQ.NET does not make any properties available for applications to use to set this value. Instead, IBM MQ uses a mapping table to internally map the CipherSpec set to the Protocol family and identifies the SSLProtocol version to be used. This table shows the mapping each of the supported CipherSpec between Microsoft.NET and IBM MQ, and the Protocol version to which they belong. For more information, see “CipherSpec mappings for the managed .NET client” on page 580.

CipherSpec support for the managed .NET client
The CipherSpec settings for an application are used during the handshake with the server.

IBM MQ clients allow you to set a CipherSpec value that is used during the handshake with the queue manager. IBM MQ clients should set a valid CipherSpec for secured connection to establish, preferably the CipherSpec specified in the Windows group policy. Leaving this field blank indicates a plain-text channel without any security on the sockets.

For the IBM MQ.NET managed client, the TLS settings are for the Microsoft.NET SSLStream class. For SSLStream, a CipherSpec, or a preference list of CipherSpecs, can be set only in the Windows group policy, which is a computer-wide setting. SSLStream then uses the specified CipherSpec or preference list during the handshake with the server. In case of other IBM MQ clients, the CipherSpec property can be set in the application on the IBM MQ channel definition and the same setting is used for TLS negotiation. As a result of this restriction, the TLS handshake might negotiate any supported CipherSpec regardless of what is specified in the IBM MQ channel configuration. Therefore, it is likely that this will result in error AMQ9631 on the queue manager. To avoid this error, set the same CipherSpec as the one that you have set in the application as the TLS configuration in the Windows group policy.

The new IBM MQ.NET TLS client code checks only that the correct protocol version was negotiated. The TLS protocol version is derived from the CipherSpec that the application sets and is used for the TLS handshake with the server (queue manager). Hence it is required by design to set the CipherSpec in the IBM MQ.NET managed client application. If the CipherSpec set by the IBM MQ client is anything other than the one from the SSL 3.0, TLS 1.0 and TLS 1.2 protocols, the IBM MQ managed .NET client would negotiate by default with any of the ciphers from SSL 3.0 or TLS 1.0 protocols and it would not report an error.

Note: If the CipherSpec value supplied by the application is not a CipherSpec known to IBM MQ, then the IBM MQ managed .NET client disregards it and negotiates the connection based on the Windows system's group policy.

Setting a CipherSpec

There are three ways of setting a CipherSpec:

MQEnvironment .NET class

The following example shows how to set a CipherSpec with the MQEnvironment class.

```
MQEnvironment.SSLKeyRepository = "*USER";
MQEnvironment.ConnectionName = connectionName;
MQEnvironment.Channel = channelName;
MQEnvironment.properties.Add(MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES_MANAGED);
MQEnvironment.SSLCipherSpec = "TLS_RSA_WITH_AES_128_CBC_SHA";
```

TLS CipherSpec property

The following example shows how to set a CipherSpec by adding a hashtable parameter into the MQQueueManager constructor.

```
properties = new Hashtable();
properties.Add(MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES_MANAGED);
properties.Add(MQC.HOST_NAME_PROPERTY, hostName);
properties.Add(MQC.PORT_PROPERTY, port);
properties.Add(MQC.CHANNEL_PROPERTY, channelName);
properties.Add(MQC.SSL_CERT_STORE_PROPERTY, sslKeyRepository);
properties.Add(MQC.SSL_CIPHER_SPEC_PROPERTY, cipherSpec);
properties.Add(MQC.SSL_PEER_NAME_PROPERTY, sslPeerName);
properties.Add(MQC.SSL_RESET_COUNT_PROPERTY, keyResetCount);
queueManager = new MQQueueManager(queueManagerName, properties);
```

Windows group policy

When a Cipher Suite list is configured via the Windows group policy management console, the SVRCONN channel definition must specify a matching CipherSpec. A matching CipherSpec could either be a generic value such as "ANY_TLS12_OR_HIGHER", or a specific value that maps to the highest Cipher Suite that would be negotiated from the ordered list. The use of generic CipherSpec values is recommended for use with .NET clients as it avoids needing to change the SVRCONN CipherSpec configuration if the order of the client list changes.

CCDT usage

IBM MQ.NET only supports Client Channel Definition Tables (.TAB files) that are on a local computer. Existing CCDT files that have a CipherSpec value set can be used for IBM MQ.NET connections. However, the CipherSpec value set on the client connection channel determines the TLS protocol version and also must match the CipherSpec set in the Windows group policy.

Related concepts

“Setting up the IBM MQ environment” on page 566
Before you use the client connection to connect to a queue manager, you must set up the IBM MQ environment.

“TLS support for the managed .NET client” on page 576
The managed .NET client uses the Microsoft .NET Framework libraries to implement TLS secure socket protocols. The Microsoft System.Net.SecuritySslStream class operates as a stream over connected TCP sockets and sends and receives data over that socket connection.

Related tasks

Specifying CipherSpecs

Related reference

MQEnvironment .NET class

CipherSpec mappings for the managed .NET client
The IBM MQ.NET interface maintains an IBM MQ to Microsoft.NET mapping table that is used to determine the version of the TLS protocol that the managed client needs to use to establish a secure connection with a queue manager.

If a CipherSpec is specified on the SVRCONN channel, then after the TLS handshake is complete, the queue manager tries to match that CipherSpec with the negotiated CipherSpec that the client application is using. If the queue manager cannot find a matching CipherSpec, the communication fails with error AMQ9631.

The IBM MQ.NET interface maintains an IBM MQ to Microsoft.NET CipherSpec mapping table. This table is used to determine the TLS protocol version that client wants to use to establish a secured socket connection with the queue manager. Based on the SSLCipherSpec value, the SSLProtocol version can be TLS 1.0, or TLS 1.2, depending on which version of the Microsoft.NET Framework you are using.

Make sure that you provide the correct the SSLCipherSpec value as specifying an incorrect value might result in SSL 3.0 or TLS 1.0 protocols being used.

Table 78. IBM MQ and Microsoft.NET mapping table

IBM MQ CipherSpec	**Microsoft.NET CipherSpec**	**TLS version**
TLS_RSA_WITH_AES_128_CBC_SHA	TLS_RSA_WITH_AES_128_CBC_SHA	TLS 1.0
TLS_RSA_WITH_AES_256_CBC_SHA	TLS_RSA_WITH_AES_256_CBC_SHA	TLS 1.0
TLS_RSA_WITH_3DES_EDE_CBC_SHA [1]	TLS_RSA_WITH_3DES_EDE_CBC_SHA [1]	TLS 1.0
TLS_RSA_WITH_AES_128_CBC_SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256	TLS 1.2
TLS_RSA_WITH_AES_256_CBC_SHA256	TLS_RSA_WITH_AES_256_CBC_SHA256	TLS 1.2
ECDHE_RSA_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256_P256	TLS 1.2
ECDHE_RSA_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256_P384	TLS 1.2
ECDHE_RSA_AES_128_CBC_SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256_P521	TLS 1.2
ECDHE_ECDSA_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256_P256	TLS 1.2
ECDHE_ECDSA_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256_P384	TLS 1.2
ECDHE_ECDSA_AES_128_CBC_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256_P521	TLS 1.2
ECDHE_ECDSA_AES_256_CBC_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384_P384	TLS 1.2
ECDHE_ECDSA_AES_256_CBC_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384_P521	TLS 1.2
ECDHE_RSA_AES_128_GCM_SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS 1.2
ECDHE_RSA_AES_256_GCM_SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS 1.2

Table 78. IBM MQ and Microsoft.NET mapping table (continued)		
IBM MQ CipherSpec	**Microsoft.NET CipherSpec**	**TLS version**
ECDHE_ECDSA_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256_P256	TLS 1.2
ECDHE_ECDSA_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256_P384	TLS 1.2
ECDHE_ECDSA_AES_128_GCM_SHA256	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256_P521	TLS 1.2
ECDHE_ECDSA_AES_256_GCM_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384_P384	TLS 1.2
ECDHE_ECDSA_AES_256_GCM_SHA384	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384_P521	TLS 1.2
V 9.4.0 TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384	TLS 1.3
V 9.4.0 TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256	TLS 1.3
V 9.4.0 TLS_AES_128_GCM_SHA256	TLS_AES_128_GCM_SHA256	TLS 1.3
V 9.4.0 TLS_AES_128_CCM_8_SHA256	TLS_AES_128_CCM_8_SHA256	TLS 1.3
V 9.4.0 TLS_AES_128_CCM_SHA256	TLS_AES_128_CCM_SHA256	TLS 1.3

Notes:

1. Deprecated This CipherSpec TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.

Related concepts

“TLS support for the managed .NET client” on page 576
The managed .NET client uses the Microsoft .NET Framework libraries to implement TLS secure socket protocols. The Microsoft System.Net.SecuritySslStream class operates as a stream over connected TCP sockets and sends and receives data over that socket connection.

Key repositories for the managed .NET client
The key repository used by managed .NET clients is the Windows keystore. Certificates and private keys must be available in either the user or system keystore to be able to be used by the client application for both identity and trust during a TLS handshake.

Client side

In the application, you can set either of the following values for the key repository:

- "*USER": IBM MQ.NET accesses the current user's certificate store to retrieve the client certificates.
- "*SYSTEM": IBM MQ.NET accesses the local computer account to retrieve the certificates.

The client's certificates must be stored in the My certificate store of the user or computer account. All the server (CA) certificates must be stored in the root directory of the certificate store.

Note: You can store more than one certificate in a single file in the following formats:

- Personal Information Exchange - PKCS #12 (.PFX, .P12)
- Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
- Microsoft Serialized Certificate Store (.SST)

Using certificates for the managed .NET client
For client certificates, the IBM MQ managed .NET client accesses the Windows keystore and loads all of the client's certificates that are matched either by certificate label or matched by the string.

When selecting a certificate to use, the IBM MQ managed .NET client always uses the first matching certificate for the SSLStream TLS handshake.

Matching certificates by certificate label

If you set the certificate label, the IBM MQ managed .NET client searches the Windows certificate store with the given label name to identify the client certificate. It loads all matching certificates and uses the first certificate on the list. There are two options for setting the certificate label:

- The certificate label can set on the MQEnvironment class accessing MQEnvironment.CertificateLabel.
- The certificate label can also be set in a hash table properties, supplied as input parameter with MQQueueManager constructor as shown in the following example.

```
Hashtable properties = new Hashtable();
properties.Add("CertificateLabel", "mycert");
```

The name("CertificateLabel") and the value are case sensitive.

Matching certificates by string

If certificate label is not set, then the certificate that matches the string "`ibmwebspheremq`" and the current logged on user (in lowercase) is searched for and used.

Related tasks
Connecting a client to a queue manager securely
Related reference
MQEnvironment .NET class

SSLPEERNAME
The SSLPEERNAME attribute is used to check the Distinguished Name (DN) of the certificate from the peer queue manager.

In IBM MQ.NET, applications can use SSLPEERNAME to specify a distinguished name pattern as shown in the following example.

```
SSLPEERNAME(CN=QMGR.*, OU=IBM, OU=WEBSPHERE)
```

As for other IBM MQ clients, SSLPEERNAME is an optional parameter.

If SSLPEERNAME value is not set, the IBM MQ.NET managed client does not do any Remote(Server) certificate validation and the managed client just accepts the Remote(/server) certificate as-is.

The way in which you set SSLPEERNAME depends on which of the IBM MQ stack offerings you are using.

IBM MQ classes for .NET
There are three options as follows.

1. Set MQEnvironment.SSLPeerName in the MQEnvironment class.

2. `MQEnvironment.properties.Add(MQC.SSL_PEER_NAME_PROPERTY, value)`
3. Use the queue manager constructor `MQQueueManager (String queueManagerName, Hashtable properties)`. Supply the SSLPEERNAME in the `Hashtable properties` as for option 2.

XMS .NET

Set the SSL peer name in the connection factory:

```
ConnectionFactory.SetStringProperty(XMSC.WMQ_SSL_PEER_NAME, value);
```

WCF

Include SslPeerName as a semicolon separated field in the URI.

Related reference

MQEnvironment .NET class

Secret key reset or renegotiation for the managed .NET client

The SSLStream class does not support secret key reset/renegotiation. However, to be consistent with other IBM MQ clients, the IBM MQ managed .NET client allows applications to set **SSLKeyResetCount**.

When the limit is reached, IBM MQ.NET disconnects from the queue manager and application are notified of that as an exception with MQRC_CONNECTION_BROKEN as the reason code. Applications can choose to handle the exception and re-establish connections or enable the MQCNO_RECONNECT option for IBM MQ.NET to automatically reconnect to the queue manager.

Enabling the automatic client reconnection facility means that, when the key reset count is reached, all existing connections are brought down and the IBM MQ.NET client re-creates all the connections afresh. For more information about automatic client reconnection, see Automatic client reconnection.

Related concepts

Resetting SSL and TLS secret keys

Revocation check

The SSLStream class supports certificate revocation checking.

The revocation checking is automatically done by the certificate chaining engine. This applies for both Online Certificate Status Protocol (OCSP) and Certificate Revocation lists (CRLs). The SSLStream class uses the certificate revocation that uses only the server specified in the certificate, that is the server is dictated by the certificate itself. It is possible for HTTP CDP extensions and OCSP HTTP requests to proxy through HTTP proxy server.

The way in which you set the revocation check depends on which of the IBM MQ stack offerings you are using.

IBM MQ.NET

The revocation check can be set by accessing the **MQEnvironment.SSLCertRevocationCheck** property on the `MQEnvironment.cs` class file.

XMS .NET

The revocation check can be set on the connection factory property context as shown in the following example.

```
ConnectionFactory.SetBooleanProperty(XMSC.WMQ_SSL_CERT_REVOCATION_CHECK, true);
```

WCF

The revocation check can be set on the URI using the following naming convention.

```
"SslCertRevocationCheck=true"
```

Configuring TLS for managed IBM MQ .NET
Configuring TLS for managed IBM MQ .NET consists of creating the signer certificates, then configuring the server side, the client side, and the application program.

About this task

To configure TLS, you must first create the appropriate signer certificates. Signer certificates can be either self signed or certificates provided by a certificate authority. Although self-signed certificates can be used on a development, test or pre-production system, do not use them on a production system. On a production system, use certificates that you have obtained from a trusted external certificate authority (CA).

Procedure

1. Create the signer certificates.
   a) To create self-signed certificates, use the **runmqakm** or V 9.4.0 V 9.4.0 **runmqktool** commands.
      For more information, see Creating a self-signed personal certificate on AIX, Linux, and Windows.
   b) To obtain certificates for the queue manager and clients from a certificate authority (CA), follow the instructions in Obtaining personal certificates from a certificate authority.
2. Configure the server side.
   a) Configure TLS on the queue manager, using IBM Global Security Kit (GSKit), as described in Connecting a client to a queue manager securely.
   b) Set the SVRCONN channel TLS attributes:
      - Set **SSLCAUTH** to REQUIRED or OPTIONAL.
      - Set **SSLCIPH** to an appropriate CipherSpec.

      For more information, see "Enabling TLS for the unmanaged .NET client" on page 575.
3. Configure the client side.
   a) Import the client certificates into the Windows certificate store (under the User/Computer account).
      IBM MQ .NET accesses client certificates from the Windows certificate store, therefore you must import your certificates into the Windows certificate store to establish a secure socket connection to IBM MQ . For more information about how to access the Windows keystore and import the client side certificates, see Import or export certificates and private keys.
   b) Supply the CertificateLabel as described in Connecting a client to a queue manager securely.
   c) If needed, edit the Windows Group Policy to set the CipherSpec, then, for the Windows Group Policy updates to take effect, restart the computer.
4. Configure the application program.
   a) Set the MQEnvironment or the SSLCipherSpec value to denote the connection as a secured connection.
      The value that you specify is used to identify the protocol being used (TLS). The CipherSpec set should be one of the CipherSpecs of the supported SSLProtocol version and it can preferably be the same as the one specified in the Windows Group Policy. (The supported SSLProtocol version depends on the .NET framework used. The SSLProtocol version can be TLS 1.0, or TLS 1.2, depending on which version of the Microsoft .NET Framework you are using.)

      Note: If the CipherSpec value supplied by the application is not a CipherSpec known to IBM MQ, then the IBM MQ managed .NET client disregards it and negotiates the connection based on the Windows system's group policy.
   b) Set the SSLKeyRepository property to either "*SYSTEM" or "*USER".
   c) Optional: Set SSLPEERNAME to the distinguished name (DN) of the server certificate.
   d) Supply the CertificateLabel as described in Connecting a client to a queue manager securely.

e) Set any further optional parameters that you require such as KeyResetCount, CertificationRevocationCheck, and enable FIPS.

Examples of how to set the TLS protocol and TLS key repository

For Base .NET, you can set the TLS protocol and TLS key repository through the MQEnvironment class as shown in the following example:

```
MQEnvironment.SSLCipherSpec = "TLS_RSA_WITH_AES_128_CBC_SHA256";
MQEnvironment.SSLKeyRepository = "*USER";

MQEnvironment.properties.Add(MQC.SSL_CIPHER_SPEC_PROPERTY, "TLS_RSA_WITH_AES_128_CBC_SHA256")
```

Alternatively, you can set the TLS protocol and TLS key repository by supplying a hashtable as part of the MQQueueManager constructor as shown in the following example.

```
Hashtable properties = new Hashtable();
properties.Add(MQC.SSL_CERT_STORE_PROPERTY, sslKeyRepository);
properties.Add(MQC.SSL_CIPHER_SPEC_PROPERTY, "TLS_RSA_WITH_AES_128_CBC_SHA256")
```

What to do next

For more information about getting started with developing IBM MQ .NET managed TLS applications, see“Writing a simple application” on page 585.

Related reference

MQEnvironment .NET class

KeyResetCount (MQLONG)

Federal Information Processing Standards (FIPS) for AIX, Linux, and Windows

Writing a simple application
Tips for writing a simple IBM MQ managed .NET TLS application, including examples for setting the SSL properties for connection factories, creating a queue manager instance, connection, session and destination, and sending a test message.

Before you begin

You must first configure TLS for managed IBM MQ.NET as described in “Configuring TLS for managed IBM MQ .NET” on page 584.

For application program configuration in base .NET, set SSL properties either using the MQEnvironment class or by supplying a hashtable as part of the MQQueueManager constructor.

For application program configuration in XMS .NET, you set the SSL properties on the property context of the connection factories.

Procedure

1. Set the SSL properties for the connection factories as shown in the following examples.

   Example for IBM MQ.NET

```
properties = new Hashtable();
properties.Add(MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES_MANAGED);
properties.Add(MQC.HOST_NAME_PROPERTY, hostName);
properties.Add(MQC.PORT_PROPERTY, port);
properties.Add(MQC.CHANNEL_PROPERTY, channelName);
properties.Add(MQC.SSL_CERT_STORE_PROPERTY, sslKeyRepository);
properties.Add(MQC.SSL_CIPHER_SPEC_PROPERTY, cipherSpec);
properties.Add(MQC.SSL_PEER_NAME_PROPERTY, sslPeerName);
properties.Add(MQC.SSL_RESET_COUNT_PROPERTY, keyResetCount);
```

```
properties.Add("CertificateLabel", "ibmwebspheremq");
MQEnvironment.SSLCertRevocationCheck = sslCertRevocationCheck;
```

Example for XMS .NET

```
cf.SetStringProperty(XMSC.WMQ_SSL_KEY_REPOSITORY, "sslKeyRepository");
cf.SetStringProperty(XMSC.WMQ_SSL_CIPHER_SPEC, cipherSpec);
cf.SetStringProperty(XMSC.WMQ_SSL_PEER_NAME, sslPeerName);
cf.SetIntProperty(XMSC.WMQ_SSL_KEY_RESETCOUNT, keyResetCount);
cf.SetBooleanProperty(XMSC.WMQ_SSL_CERT_REVOCATION_CHECK, true);
```

2. Create the queue manager instance, connections, session and destination as shown in the following examples.

   Example for MQ .NET

   ```
   queueManager = new MQQueueManager(queueManagerName, properties);
   Console.WriteLine("done");

   // accessing queue
   Console.Write("Accessing queue " + queueName + ".. ");
   queue = queueManager.AccessQueue(queueName, MQC.MQOO_OUTPUT +
   MQC.MQOO_FAIL_IF_QUIESCING);
   Console.WriteLine("done");
   ```

   Example for XMS .NET

   ```
   connectionWMQ = cf.CreateConnection();
   // Create session
   sessionWMQ = connectionWMQ.CreateSession(false, AcknowledgeMode.AutoAcknowledge);

   // Create destination
   destination = sessionWMQ.CreateQueue(destinationName);

   // Create producer
   producer = sessionWMQ.CreateProducer(destination);
   ```

3. Send a message as shown in the following examples.

   Example for MQ .NET

   ```
   // creating a message object
   message = new MQMessage();
   message.WriteString(messageString);

   // putting messages continuously
   for (int i = 1; i <= numberOfMsgs; i++)
   {
   Console.Write("Message " + i + " <" + messageString + ">.. ");
   queue.Put(message);
   Console.WriteLine("put");
   }
   ```

   Example for XMS .NET

   ```
   textMessage = sessionWMQ.CreateTextMessage();
   textMessage.Text = simpleMessage;
   producer.Send(textMessage);
   ```

4. Verify the TLS connection.

   Check the channel status to verify that the TLS connection has been established and is working correctly.

Configuring trace for SSLStream
To capture tracing events and messages relating to the SSLStream class, you must add a configuration section for system diagnostics to the application configuration file for your application.

About this task

Note:

This task applies to IBM MQ classes for .NET Framework only. The application configuration file is not supported in IBM MQ classes for .NET (.NET Standard and .NET 6 libraries).

If you do not add a configuration section for system diagnostics to the application configuration file, the IBM MQ managed .NET client will not capture any events, traces or debugging points relating to TLS and the SSLStream class.

Note: Starting IBM MQ tracing using **strmqtrc** does not capture all the required TLS tracing.

Procedure

1. Create an application configuration (App.Config) file for your application project.
2. Add a system diagnostics configuration section as shown in the following example.

```
<system.diagnostics>
           <sources>
                 <source name="System.Net" tracemode="includehex">
                       <listeners>
                             <add name="ExternalSourceTrace"/>
                       </listeners>
                 </source>
                 <source name="System.Net.Sockets">
                       <listeners>
                             <add name="ExternalSourceTrace"/>
                       </listeners>
                 </source>
                 <source name="System.Net.Cache">
                       <listeners>
                             <add name="ExternalSourceTrace"/>
                       </listeners>
                 </source>
                 <source name="System.Net.Security">
                       <listeners>
                             <add name="ExternalSourceTrace"/>
                       </listeners>
                 </source>
                 <source name="System.Security">
                       <listeners>
                             <add name="ExternalSourceTrace"/>
                       </listeners>
                 </source>
           </sources>
           <switches>
                 <add name="System.Net" value="Verbose"/>
                 <add name="System.Net.Sockets" value="Verbose"/>
                 <add name="System.Net.Cache" value="Verbose"/>
                 <add name="System.Security" value="Verbose"/>
                 <add name="System.Net.Security" value="Verbose"/>
           </switches>

           <sharedListeners>
                 <add name="ExternalSourceTrace" type="IBM.WMQ.ExternalSourceTrace,
amqmdnet, Version=n.n.n.n, Culture=neutral, PublicKeyToken=dd3cb1c9aae9ec97" />
           </sharedListeners>
         <trace autoflush="true"/>
     </system.diagnostics>
```

Attention: The `Version` field of the add `name` entry needs to be whichever version of the .net `amqmdnet.dll` file that is being used.

Related tasks

Tracing IBM MQ classes for .NET Framework clients using an application configuration file

Sample applications for implementing TLS in managed .NET
Sample applications are provided to show the implementation of TLS for managed .NET in IBM MQ classes for .NET, XMS .NET and IBM MQ custom channel for WCF.

The following table shows the location of the sample applications. *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

Table 79. Location of sample applications for implementing TLS in managed .NET

IBM MQ.NET stack offering	**Location of samples**
Base .NET	*MQ_INSTALLATION_PATH*\Tools\dotnet\samples\cs\base\SimplePut\SimplePut.cs *MQ_INSTALLATION_PATH*\Tools\dotnet\samples\cs\base\SimpleGet\SimpleGet.cs
XMS .NET	*MQ_INSTALLATION_PATH*\Tools\dotnet\samples\cs\xms\simple\wmq\SimpleProducer\SimpleProducer.cs *MQ_INSTALLATION_PATH*\Tools\dotnet\samples\cs\xms\simple\wmq\SimpleConsumer\SimpleConsumer.cs
IBM MQ custom channel for WCF	*MQ_INSTALLATION_PATH*\Tools\dotnet\samples\cs\wcf\samples\WCF\oneway\service\MQMessagingOneWayService.cs

Windows Using the .NET Monitor

The .NET Monitor is an application similar to an IBM MQ trigger monitor.

Important: See Features that can be used only with the primary installation on Windows for important information.

You can create .NET components which are instantiated whenever a message is received on a monitored queue, and which then process that message. The .NET Monitor is started by the **runmqdnm** command and stopped by the **endmqdnm** command. For details of these commands, see runmqdnm and endmqdnm.

To use the .NET Monitor, you write a component that implements the IMQObjectTrigger interface, which is defined in amqmdnm.dll.

Components can be either transactional or non-transactional. A transactional component must inherit from System.EnterpriseServices.ServicedComponent and be registered as either RequiresTransaction or SupportsTransaction. It must not be registered as RequiresNew as the .NET Monitor has already initiated a transaction.

The component receives MQQueueManager, MQQueue, and MQMessage objects from **runmqdnm**. It may also receive a User Parameter string if one was specified, using the *-u* command-line option, when runmqdnm was started. Note that your component receives the contents of a message that arrived on the monitored queue in an MQMessage object. It does not have to connect to the queue manager, open the queue, or get the message itself. The component must then process the message as appropriate and return control to the .NET Monitor.

If your component has been written as a transactional component, it registers to commit or roll back the transaction using the facilities provided by System.EnterpriseServices.ServicedComponent.

As the component receives MQQueueManager and MQQueue objects as well as the message, it has complete context information for that message and can, for example, open another queue on the same queue manager without needing to separately connect to IBM MQ.

Windows *Example code fragments*

This topic contains two examples of components which obtain a message from the .NET Monitor and print it, one using transactional processing and the other non-transactional processing. A third example shows common utility routines, applicable to both the first two examples. All the examples are in C#.

Example 1: Transactional processing

```
/*********************************************************************/
/* Licensed materials, property of IBM                               */
/* 63H9336                                                           */
/* (C) Copyright IBM Corp. 2005, 2024.                               */
/*********************************************************************/
using System;
using System.EnterpriseServices;

using IBM.WMQ;
using IBM.WMQMonitor;

[assembly: ApplicationName("dnmsamp")]

// build:
//
// csc -target:library -reference:amqmdnet.dll;amqmdnm.dll TranAssembly.cs
//
// run (with dotnet monitor)
//
// runmqdnm -m QMNAME -q QNAME -a dnmsamp.dll -c Tran

namespace dnmsamp
{
  [TransactionAttribute(TransactionOption.Required)]
  public class Tran : ServicedComponent, IMQObjectTrigger
  {
    Util util = null;

    [AutoComplete(true)]
    public void Execute(MQQueueManager qmgr, MQQueue queue,
        MQMessage message, string param)
    {
      util = new Util("Tran");

      if (param != null)
        util.Print("PARAM: '" +param.ToString() + "'");

      util.PrintMessage(message);

      //System.Console.WriteLine("SETTING ABORT");
      //ContextUtil.MyTransactionVote = TransactionVote.Abort;

      System.Console.WriteLine("SETTING COMMIT");
      ContextUtil.SetComplete();
      //ContextUtil.MyTransactionVote = TransactionVote.Commit;
    }
  }
}
```

Example 2: Non-transactional processing

```
/*********************************************************************/
/* Licensed materials, property of IBM                               */
/* 63H9336                                                           */
/* (C) Copyright IBM Corp. 2005, 2024.                               */
/*********************************************************************/

using System;

using IBM.WMQ;
using IBM.WMQMonitor;

// build:
//
// csc -target:library -reference:amqmdnet.dll;amqmdnm.dll NonTranAssembly.cs
//
// run (with dotnet monitor)
```

```
//
// runmqdnm -m QMNAME -q QNAME -a dnmsamp.dll -c NonTran
namespace dnmsamp
{
  public class NonTran : IMQObjectTrigger
  {
    Util util = null;

    public void Execute(MQQueueManager qmgr, MQQueue queue,
        MQMessage message, string param)
    {
      util = new Util("NonTran");

      try
      {
        util.PrintMessage(message);
      }

      catch (Exception ex)
      {
        System.Console.WriteLine(">>> NonTran\n{0}", ex.ToString());
      }
    }
  }
}
```

Example 3: Common routines

```
/*********************************************************************/
/* Licensed materials, property of IBM                               */
/* 63H9336                                                           */
/* (C) Copyright IBM Corp. 2005, 2024.                               */
/*********************************************************************/

using System;

using IBM.WMQ;

namespace dnmsamp
{
    /// <summary>
    /// Summary description for Util.
    /// </summary>
    public class Util
    {
    /* ---------------------------------------------------------------- */
    /* Default prefix string of the namespace.                           */
    /* ---------------------------------------------------------------- */
    private string prefixText = "dnmsamp";

    /* ---------------------------------------------------------------- */
    /* Constructor that takes the replacement prefix string to use.     */
    /* ---------------------------------------------------------------- */
    public Util(String text)
    {
      prefixText = text;
    }

    /* ---------------------------------------------------------------- */
    /* Display an arbitrary string to the console.                      */
    /* ---------------------------------------------------------------- */
    public void Print(String text)
    {
      System.Console.WriteLine("{0} {1}\n", prefixText, text);
    }



 /* ---------------------------------------------------------------- */
    /* Display the content of the message passed to the console.        */
    /* ---------------------------------------------------------------- */
    public void PrintMessage(MQMessage message)
    {
      if (message.Format.CompareTo(MQC.MQFMT_STRING) == 0)
      {
        try
        {
          string messageText = message.ReadString(message.MessageLength);
```

```
          Print(messageText);
        }

        catch(Exception ex)
        {
          Print(ex.ToString());
        }
      }
      else
      {
        Print("UNRECOGNISED FORMAT");
      }
    }

    /* ------------------------------------------------------------------ */
    /* Convert the byte array into a hex string.                          */
    /* ------------------------------------------------------------------ */
    static public string ToHexString(byte[] byteArray)
    {
      string hex = "0123456789ABCDEF";

      string retString = "";

      for(int i = 0; i < byteArray.Length; i++)
      {
        int h = (byteArray[i] & 0xF0)>>4;
        int l = (byteArray[i] & 0x0F);

        retString += hex.Substring(h,1) + hex.Substring(l,1);
      }

      return retString;
    }
  }
}
```

Compiling IBM MQ .NET programs

Specimen commands to compile .NET applications written in various languages.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

To build a C# application using IBM MQ classes for .NET, use the following command:

```
csc /t:exe /r:System.dll /r:amqmdnet.dll /lib: MQ_INSTALLATION_PATH\bin /out:MyProg.exe
MyProg.cs
```

To build a Visual Basic application using IBM MQ classes for .NET, use the following command:

```
vbc /r:System.dll /r: MQ_INSTALLATION_PATH\bin\amqmdnet.dll /out:MyProg.exe MyProg.vb
```

To build a Managed C++ application using IBM MQ classes for .NET, use the following command:

```
cl /clr MQ_INSTALLATION_PATH\bin Myprog.cpp
```

For other languages, see the documentation supplied by the language vendor.

Using the stand-alone IBM MQ .NET client

The IBM MQ .NET client offers you the ability to package and deploy an IBM MQ .NET assembly without needing to use the full IBM MQ client installation on production systems for running your applications.

Before you begin

V 9.4.0 From IBM MQ 9.4.0, the `amqmdnetstd.dll` client library installed in the default location is based on .NET 6.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for .NET. If you are using a .NET 6 application, you can run this application without any recompilation being required by making a small edit in the `runtimeconfig` file to set the `targetframeworkversion` to `"net8.0"`.

Removed V 9.4.0 V 9.4.0 The IBM MQ .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product IBM MQ 9.4.0.

Stabilized LTS The `amqmdnet.dll` library is still supplied, but this library is stabilized; that is, no new features will be introduced into it. For any of the latest features you must migrate to the `amqmdnetstd.dll` library. However, you can continue to use the `amqmdnet.dll` library on IBM MQ 9.1 Long Term Support or Continuous Delivery releases.

About this task

You can build your IBM MQ .NET applications on a machine where the full IBM MQ client is installed and later package the IBM MQ .NET assembly, that is, `amqmdnetstd.dll`, along with your application and deploy it on production systems.

The applications that you build and deploy can be the traditional .NET applications, Services, or Microsoft Azure Web/Worker applications

In such deployments, the IBM MQ .NET client supports only the managed mode of connectivity to a queue manager. The server bindings and unmanaged client mode connectivity are not available as these two modes require a full IBM MQ client installation. Any attempt to use these other two modes results in an application exception.

Procedure

Referencing the IBM MQ .NET client assembly in applications

- Reference the `amqmdnetstd.dll` assembly in your application in the same way that you did for earlier releases.

  Set the **CopyLocal** property of the `amqmdnetstd.dll` assembly to `True` to ensure that the `amqmdnetstd.dll` assembly is copied to the `bin` directory of the application. Setting this property also helps the application packaging tool to package the required binary files for deployment on production systems as well as Microsoft Azure PaaS cloud environments.

Adding global transaction support

- Ensure that your application deploys the monitor application WMQDotnetXAMonitor on the machine along with the application itself.

  If an application uses the IBM MQ .NET managed global transaction feature, then it must also deploy the WMQDotnetXAMonitor on the machine along with the application itself. This utility is needed for recovering any in-doubt transactions.

Starting and stopping trace

- For IBM MQ classes for .NET Framework only, to start and stop trace using the application configuration file and an IBM MQ specific trace configuration file, see Tracing an IBM MQ classes for .NET Framework client using an application configuration file.

  You must use the application configuration file and an IBM MQ specific trace configuration file because, since there is no full IBM MQ client installation, the standard tools that are used for starting and stopping trace, **strmqtrc** and **endmqtrc**, are not available.

  Notes:

  - This way of generating trace applies to the .NET redistributable managed client as well as the stand-alone .NET client. See .NET application runtime - Windows only.
  - The application configuration file is not supported in IBM MQ classes for .NET (.NET Standard and .NET 6 libraries). To enable trace for IBM MQ classes for .NET (.NET Standard and .NET

6 libraries), you use the **MQDOTNET_TRACE_ON** environment variable. See Tracing IBM MQ .NET applications using environment variables.

- V 9.4.0

  Start and stop trace by using the `mqclient.ini` file and setting the appropriate properties of the Trace stanza.

  See Tracing IBM MQ .NET applications with mqclient.ini.

  From IBM MQ 9.4.0, you can configure trace by using the `mqclient.ini` file and setting the appropriate properties of the Trace stanza. You can also enable and disable tracing dynamically with the `mqclient.ini` file.

Enabling binding redirection in the application configuration file

- To enable compile time binding reference of the IBM MQ .NET assembly to a later version of the assembly, add the <dependentAssembly> property to the application configuration file.

  The following example snippet in the `app.config` file redirects an application that was compiled using the IBM MQ 8.0.0 Fix Pack 2 (`8.0.0.2`) version of the IBM MQ .NET assembly but later a fix pack, IBM MQ 8.0.0 Fix Pack 3, was then applied that updated IBM MQ.NET assembly to `8.0.0.3`.

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <!-- amqmdnet related binding redirect -->
    <dependentAssembly>
      <assemblyIdentity name="amqmdnet"
                        publicKeyToken="dd3cb1c9aae9ec97"
                        culture="neutral" />
      <codeBase version="8.0.0.2"
            href="file:///amqmdnet.dll"/>
      <bindingRedirect oldVersion="1.0.0.3-8.0.0.2"
                       newVersion="8.0.0.3"/>
      <publisherPolicy apply="no" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

Related concepts

“Installing IBM MQ classes for .NET” on page 537
IBM MQ classes for .NET, including samples, are installed with IBM MQ on Windows and Linux

Redistributable clients

.NET application runtime - Windows only

Related tasks

“Using the WMQDotnetXAMonitor application” on page 556
The IBM MQ .NET client provides an XA Monitor application, WmqDotnetXAMonitor, that you can use to recover any incomplete distributed transactions. The WmqDotnetXAMonitor application establishes a connection to the queue manager where the transactions are in-doubt and then resolves the transaction based on the parameters that you set.

Tracing IBM MQ .NET applications

OutboundSNI property

You can set the **OutboundSNI** property in an application by using either a property or an environment variable.

From IBM MQ 9.3.0, you can set the MQC.OUTBOUND_SNI_PROPERTY in the application, using a hash table when using the `MQQueueManager` class to connect to the queue manager.

The MQC.OUTBOUND_SNI_PROPERTY takes the following values:

- MQC.OUTBOUND_SNI_CHANNEL, which maps to "CHANNEL"
- MQC.OUTBOUND_SNI_HOSTNAME, which maps to "HOSTNAME"
- MQC.OUTBOUND_SNI_ASTERISK, which maps to "*"

Additionally, you can set the **OutboundSNI** property using the MQOUTBOUND_SNI environment variable, which takes the following values:

- CHANNEL
- HOSTNAME
- *

and set the **OutboundSNI** value in the App.config file, as with any other mqclient.ini property.

Note: The property defaults to MQC.OUTBOUND_SNI_CHANNEL if no specific value is set.

The order of precedence for setting the **OutboundSNI** property in the managed node is:

1. Application level property
2. Environment variable

For the **OutboundSNI** property in unmanaged node, mqclient.ini only is supported.

The properties set in the App.config file are applicable for .NET Framework applications only.

If you provide a value that is not valid at the application level or in the App.config file, the return code MQRC_OUTBOUND_SNI_NOT_VALID is issued.

If you set an environment variable that is not valid, or provide a value that is not valid in the mqclient.ini file, the default value of CHANNEL is used.

OutboundSNI and multiple certificates

IBM MQ uses the SNI header to provide multiple certificates functionality. If an application is connecting to an IBM MQ channel that is configured to use a different certificate through the CERTLABL field, then the application must connect with an **OutboundSNI** setting of CHANNEL.

If an application with an **OutboundSNI** setting of anything other than CHANNEL connects to a channel with a certificate label configured, the application is rejected with an MQRC_SSL_INITIALIZATION_ERROR, and an AMQ9673 message is printed in the queue manager error logs.

For more information on how IBM MQ provides multiple certificate functionality, see How IBM MQ provides multiple certificates capability .

Developing XMS .NET applications

IBM MQ Message Service Client (XMS) for .NET (XMS .NET) provides an application programming interface (API) called XMS that has the same set of interfaces as the Java Message Service (JMS) API. IBM MQ Message Service Client (XMS) for .NET contains a fully managed implementation of XMS, which can be used by any .NET compliant language.

Before you begin

Deprecated V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, in IBM MQ classes for XMS .NET, the methods WriteObject(), ReadObject(), CreateObjectMessage(), and the classes ObjectMessage and XmsObjectMessageImpl used for serialization and deserialization of data are deprecated.

Removed V 9.4.0 V 9.4.0 The XMS .NET client library built using.NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

About this task

XMS supports:

- Point-to-point messaging
- Publish/subscribe messaging

- Synchronous message delivery
- Asynchronous message delivery

An XMS application can exchange messages with the following types of application:

- An XMS application
- An IBM MQ classes for JMS application
- A native IBM MQ application
- A JMS application that is using the IBM MQ default messaging provider

An XMS application can connect to, and use the resources of, any of the following messaging servers:

IBM MQ queue manager
The application can connect in either bindings or client mode.

WebSphere Application Server service integration bus
The application can use a direct TCP/IP connection, or it can use HTTP over TCP/IP.

IBM Integration Bus
Messages are transported between the application and the broker using WebSphere MQ Real-Time Transport. Messages can be delivered to the application using WebSphere MQ Multicast Transport.

By connecting to an IBM MQ queue manager, an XMS application can use WebSphere MQ Enterprise Transport to communicate with IBM Integration Bus. Alternatively, an XMS application can publish and subscribe by connecting to IBM MQ.

V 9.4.0 IBM MQ 9.4.0 provides an XMS .NET client library built against .NET 6 as the target framework. For more information, see “Installing IBM MQ classes for XMS .NET” on page 599.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for XMS .NET. For more information, see “Installing IBM MQ classes for XMS .NET” on page 599.

XMS .NET managed applications are able to automatically balance connections across clustered queue managers. Both the IBM MQ classes for XMS .NET and the IBM MQ classes for XMS .NET Framework libraries are supported. For more information, see About uniform clusters and Automatic application balancing.

For more information about the differences between IBM MQ classes for XMS .NET Framework and IBM MQ classes for XMS .NET, see “Installing IBM MQ classes for XMS .NET” on page 599.

Related tasks
Contacting IBM Support
Troubleshooting XMS .NETproblems problems

Styles of messaging supported by XMS

XMS supports the point-to-point and publish/subscribe styles of messaging.

Styles of messaging are also called messaging domains.

Point-to-point messaging

A common form of point-to-point messaging uses queuing. In the simplest case, an application sends a message to another application by identifying, implicitly or explicitly, a destination queue. The underlying messaging and queuing system receives the message from the sending application and routes the message to its destination queue. The receiving application can then retrieve the message from the queue.

If the underlying messaging and queuing system contains IBM Integration Bus, IBM Integration Bus might replicate a message and route copies of the message to different queues. As a result, more than one application can receive the message. IBM Integration Bus might also transform a message and add data to it.

A key characteristic of point-to-point messaging is that an application places a message onto a local queue when it sends a message. The underlying messaging and queuing system determines which destination queue the message is sent to. The receiving application retrieves the message from the destination queue.

Publish/subscribe messaging

In publish/subscribe messaging, there are two types of application: publisher and subscriber.

A *publisher* supplies information in the form of publication messages. When a publisher publishes a message, it specifies a topic, which identifies the subject of the information inside the message.

A *subscriber* is a consumer of the information that is published. A subscriber specifies the topics it is interested in by creating subscriptions.

The publish/subscribe system receives publications from publishers and subscriptions from subscribers. It routes publications to subscribers. A subscriber receives publications on only those topics to which it subscribed.

A key characteristic of publish/subscribe messaging is that a publisher identifies a topic when it publishes a message. It does not identify the subscribers. If a message is published on a topic for which there are no subscribers, no application receives the message.

An application can be both a publisher and a subscriber.

The XMS object model

The XMS API is an object-oriented interface. The XMS object model is based on the JMS 1.1 object model.

Main XMS classes

The main XMS classes, or types of object are as follows:

ConnectionFactory
A `ConnectionFactory` object encapsulates a set of parameters for a connection. An application uses a `ConnectionFactory` to create a connection. An application can provide the parameters at run time and create a `ConnectionFactory` object. Alternatively, the connection parameters can be stored in a repository of administered objects. An application can retrieve an object from the repository and create a `ConnectionFactory` object from it.

Connection
A `Connection` object encapsulates an active connection from an application to a messaging server. An application uses a connection to create sessions.

Destination
An application sends messages or receives messages using a `Destination` object. In the publish/subscribe domain, a `Destination` object encapsulates a topic and, in the point-to-point domain, a `Destination` object encapsulates a queue. An application can provide the parameters to create a `Destination` object at run time. Alternatively, it can create a `Destination` object from an object definition that is stored in repository of administered objects.

Session
A `Session` object is a single threaded context for sending and receiving messages. An application uses a `Session` object to create `Message`, `MessageProducer`, and `MessageConsumer` objects.

Message
A `Message` object encapsulates the `Message` object that an application sends using a `MessageProducer` object, or receives using a `MessageConsumer` object.

MessageProducer
A `MessageProducer` object is used by an application to send messages to a destination.

MessageConsumer
An `MessageConsumer` object is used by an application to receive messages sent to a destination.

XMS objects and their relationships

Figure 52 on page 597 shows the main types of XMS object: ConnectionFactory, Connection, Session, MessageProducer, MessageConsumer, Message, and Destination. An application uses a connection factory to create a connection, and uses a connection to create sessions. The application can then use a session to create messages, message producers, and message consumers. The application uses a message producer to send messages to a destination, and uses a message consumer to receive messages sent to a destination.

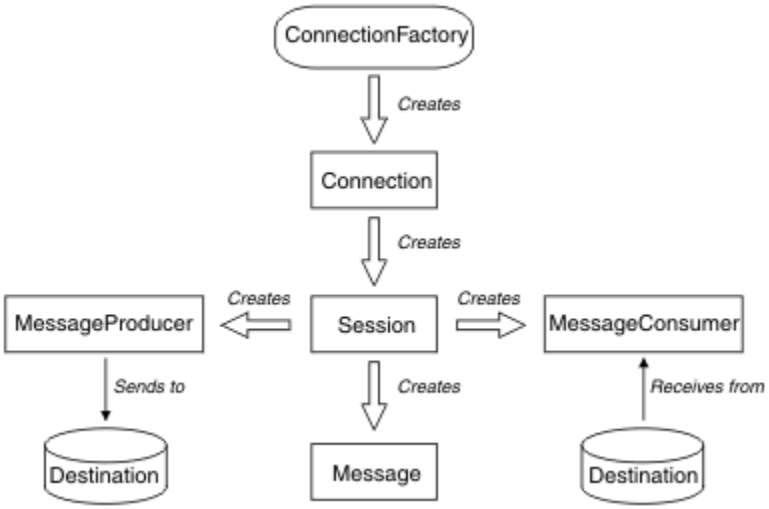


Figure 52. XMS objects and their relationships

In XMS .NET, the XMS classes are defined as a set of .NET interfaces. When you are coding XMS .NET applications, you need only the declared interfaces.

The XMS object model is based on the domain independent interfaces that are described in Java Message Service Specification, Version 1.1. Domain-specific classes, such as `Topic`, `TopicPublisher`, and `TopicSubscriber`, are not provided.

Attributes and properties of XMS objects

An XMS object can have attributes and properties, which are characteristics of the object, that are implemented in different ways:

Attributes
An object characteristic that is always present and occupies storage, even if the attribute does not have a value. In this respect, an attribute is similar to a field in a fixed-length data structure. A distinguishing feature of attributes is that each attribute has its own methods for setting and getting its value.

Properties
A property of an object is present and occupies storage only after its value is set. A property cannot be deleted or its storage recovered after its value is set. You can change its value. XMS provides a set of generic methods for setting and getting property values.

Administered objects

Using administered objects, you can administer the connection settings used by client applications to be administered from a central repository. An application retrieves object definitions from the central repository and uses them to create `ConnectionFactory` and `Destination` objects. Using administered objects, you can de-couple applications from the resources that they use at run time.

For example, XMS applications can be written and tested with administered objects that reference a set of connections and destinations in a test environment. When the applications are deployed,

the administered objects can be changed to configure the applications to refer to connections and destinations in the production environment.

XMS supports two types of administered object:

- A `ConnectionFactory` object, which is used by applications to make the initial connection to the server.
- A `Destination` object, which is used by applications to specify the destination for messages that are being sent, and the source of messages that are being received. A destination is either a topic or a queue on the server to which an application connects.

The administration tool **JMSAdmin** is supplied with IBM MQ. It is used to create and manage administered objects in a central repository of administered objects.

The administered objects in the repository can be used by IBM MQ classes for JMS and XMS applications. XMS applications can use the `ConnectionFactory` and `Destination` objects to connect to an IBM MQ queue manager. An administrator can change the object definitions held in the repository without affecting application code.

The following diagram shows how an XMS application typically uses administered objects. The left-hand side of the diagram shows a repository containing ConnectionFactory and Destination object definitions that are administered using an administration console. The right-hand side of the diagram shows an XMS application that looks up object definitions in the repository, and then uses these object definitions when connecting to a messaging server.

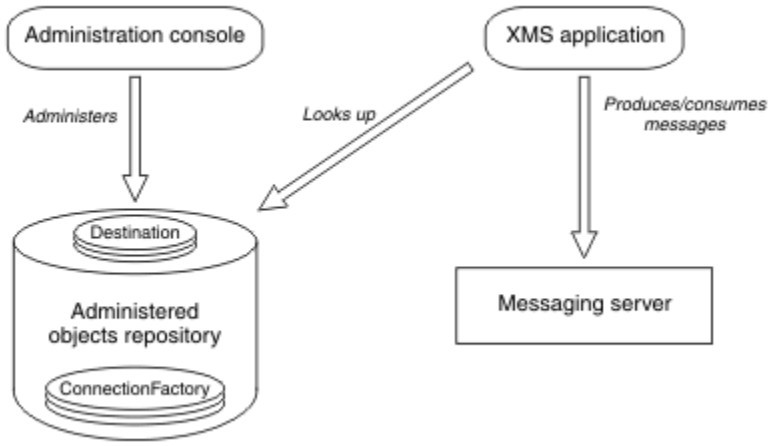


Figure 53. Typical use of administered objects by an XMS application

The XMS message model

The XMS message model is the same as the IBM MQ classes for JMS message model.

In particular, XMS implements the same message header fields and message properties that IBM MQ classes for JMS implements:

- JMS header fields. These fields have names that commence with the prefix JMS.
- JMS defined properties. These fields have properties whose names commence with the prefix JMSX.
- IBM defined properties. These fields have properties whose names commence with the prefix JMS_IBM_.

As a result, XMS applications can exchange messages with IBM MQ classes for JMS applications. In each message, some of the header fields and properties are set by the application and others are set by XMS or IBM MQ classes for JMS. Some of the fields set by XMS or IBM MQ classes for JMS are set when the message is sent, and others when it is received. Header fields and properties are propagated with a

message through a messaging server where appropriate. They are made available to any application that receives the message.

Related concepts

IBM MQ classes for JMS

Windows Linux Installing IBM MQ classes for XMS .NET

IBM MQ classes for XMS .NET, including samples, are installed with IBM MQ on Windows and Linux.

Installation

V 9.4.0 IBM MQ 9.4.0 provides an XMS .NET client library built against .NET 6 as the target framework. From IBM MQ 9.4.0, Microsoft .NET 6.0 is the minimum required version for running applications using IBM MQ libraries that are built using .NET 6 as the target framework. The XMS .NET client library built using .NET 6 as the target framework is available under *MQ_INSTALLATION_PATH*`/bin` on Windows and under *MQ_INSTALLATION_PATH*`/lib64` on Linux.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for XMS .NET. If you are using a .NET 6 application, you can run this application without any recompilation being required by making a small edit in the `runtimeconfig` file to set the `targetframeworkversion` to `"net8.0"`.

Deprecated V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, in IBM MQ classes for XMS .NET, the methods WriteObject(), ReadObject(), CreateObjectMessage(), and the classes ObjectMessage and XmsObjectMessageImpl used for serialization and deserialization of data are deprecated.

Removed V 9.4.0 V 9.4.0 The XMS .NET client library built using.NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

`amqmxmsstd.dll` library

V 9.4.0 From IBM MQ 9.4.0, the `amqmxmsstd.dll` library built using .NET 6 as the target framework is available at the following locations:

- Windows On Windows: *MQ_INSTALLATION_PATH*`\bin`. The sample applications are installed in *MQ_INSTALLATION_PATH*`/samp/dotnet/samples/cs/core/base`.
- Linux On Linux: *MQ_INSTALLATION_PATH*`\lib64`. The .NET samples are in *MQ_INSTALLATION_PATH*`/samp/dotnet/samples/cs/core/base`.

Removed V 9.4.0 V 9.4.0 The XMS .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

Attention: Removed V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, XMS .NET client libraries built using .NET Standard 2.0 as the target framework are removed. These libraries were deprecated at IBM MQ 9.3.1.

Stabilized All the `IBM.XMS.*` libraries are still supplied, but these libraries are stabilized; that is, no new features will be introduced into them. For any of the latest features, you must migrate to the `amqmxmsstd.dll` library. However, you can continue to use the existing libraries on IBM MQ 9.1 Long Term Support or Continuous Delivery releases.

V 9.4.0 V 9.4.0 Here are two scenarios that you might encounter following the removal of the `netstandard2.0` libraries:

- If you are using a IBM MQ classes for XMS .NET Framework application that is built using the `netstandard2.0` libraries such as `amqmdnetstd.dll`, you need to rebuild your application with

the Microsoft.NET Framework 4.7.2 libraries such as `amqmdnet.dll`, in order for your application to run successfully. If you do not rebuild your application, you might get an System.IO.Unexceptionable message:

```
Exception caught: System.IO.FileLoadException: Could not load file or assembly 'amqmdnetstd,
Version=9.3.5.0, Culture=neutral, PublicKeyToken=23d6cb914eeaac0e' or one of its dependencies.
The located assembly's manifest definition does not match the assembly reference. (Exception
from HRESULT: 0x80131040)
File name: 'amqmdnetstd, Version=9.3.5.0, Culture=neutral, PublicKeyToken=23d6cb914eeaac0e'
   at SimplePut.SimplePut.PutMessages()
   at SimplePut.SimplePut.Main(String[] args) in C:\SampleCode\Program.cs:line 132
```

- If you are using a .NET 6 application that is built using `netstandard2.0` libraries, then you just need to replace those libraries with the same .NET 6 libraries in the `bin` folder of the application runtime directory. No rebuild is required.

  Note: The replacement .NET 6 library should always be of the same or higher level than the replaced `netstandard2.0` library.

The IBM MQ classes for XMS .NET Standard are available for downloading from the NuGet repository. The NuGet package contains both the `amqmxmsstd.dll` library and the `amqmdnetstd.dll` library. `amqmxmsstd.dll` is dependent on `amqmdnetstd.dll` and, while packaging the XMS .NET Core application, both `amqmxmsstd.dll` and `amqmdnetstd.dll` should be packaged along with the XMS .NET Core application. For more information, see "Downloading IBM MQ classes for XMS .NET from the NuGet repository" on page 602.

dspmqver command

You can use the **dspmqver** command to display version and build information for the .NET Core component.

Comparison between IBM MQ classes for XMS .NET Framework and IBM MQ classes for XMS .NET .NET 6 libraries and .NET 6 libraries)

The following table lists the features for IBM MQ classes for XMS .NET Framework compared with the features for IBM MQ classes for XMS .NET and .NET 6 libraries).

Table 80. Differences between IBM MQ classes for XMS .NET Framework and IBM MQ classes for XMS .NET

Feature	IBM MQ classes for XMS .NET Framework	IBM MQ classes for XMS .NET
Class Names (APIs)	All classes remain the same in each network.	All classes remain the same in each network.
Operating System	Windows	Windows Dockerized containers Linux macOS
`app.config` file (Configuration file to enable Trace in redistributable client)	`app.config` file is used to enable trace for the redistributable package.	`app.config` is not supported. Use environment variables.

Table 80. Differences between IBM MQ classes for XMS .NET Framework and IBM MQ classes for XMS .NET (continued)

Feature	IBM MQ classes for XMS .NET Framework	IBM MQ classes for XMS .NET
Trace	To trace the XMS .NET client, you can use the existing environment variables, such as the environment variable **XMS_TRACE_ON** used to enable trace. For more information, see Configuring XMS .NET trace using XMS environment variables. For redistributable clients, the `app.config` file can be used to enable trace. V 9.4.0 From IBM MQ 9.4.0, you can enable and disable trace by using the `mqclient.ini` file and setting the appropriate properties of the Trace stanza. You can also enable and disable tracing dynamically with the `mqclient.ini` file. For more information, see Tracing IBM MQ .NET applications with mqclient.ini.	To trace the XMS .NET client, you can use the existing environment variables, such as the environment variable **XMS_TRACE_ON** used to enable trace. For more information, see Configuring XMS .NET trace using XMS environment variables. V 9.4.0 From IBM MQ 9.4.0, you can enable and disable trace by using the `mqclient.ini` file and setting the appropriate properties of the Trace stanza. You can also enable and disable tracing dynamically with the `mqclient.ini` file. For more information, see Tracing IBM MQ .NET applications with mqclient.ini.
Transport Modes	Managed, Unmanaged, and Bindings	Managed
TLS	The Windows keystore is used for storing the certificates.	On Windows, the keystore must be used for storing the certificates. Permitted values are *USER or *SYSTEM. Based on the input, the IBM MQ .NET client looks at the Windows key store of the current user, or System wide. On Linux, it is recommended to use the X509Store class to install certificates and .NET Core installs certificates to the following location: `".dotnet/corefx/cryptography/x509stores"`.
CCDT	Supported	Supported, and the settings of the CCDT path are the same as for .NET Framework classes.
Client auto reconnect	Supported	Supported
Distributed transactions	Supported	Not supported
Installation of dynamic linked libraries (dll's) into the global assembly cache (GAC)	Dll's are installed into the GAC as part of the IBM MQ installation.	Dll's are not installed into the GAC as part of the IBM MQ installation.
Support for WMQ, WPM, and RTT connection types	Supports WMQ, WPM, and RTT connection types	Support for WMQ only

Table 80. Differences between IBM MQ classes for XMS .NET Framework and IBM MQ classes for XMS .NET (continued)		
Feature	**IBM MQ classes for XMS .NET Framework**	**IBM MQ classes for XMS .NET**
JNDI administered objects	Supports LDAP and FileSystem	Supports FileSystem only

From IBM MQ 9.3.0, to run IBM MQ classes for XMS .NET Framework you must install Microsoft.NET Framework V4.7.2 or later.

Related tasks

"Using the XMS sample applications" on page 608
The XMS .NET sample applications provide an overview of the common features of each API. You can use them to verify your installation and messaging server setup and to help you build your own applications.

Windows Linux Downloading IBM MQ classes for XMS .NET from the NuGet repository

The IBM MQ classes for XMS .NET are available for downloading from the NuGet repository, so that they can be easily consumed by .NET Developers.

About this task

NuGet is the package manager for Microsoft development platforms including .NET. The NuGet client tools provide the ability to produce and consume packages. A NuGet package is a single compressed file with the `.nupkg` extension that contains compiled code (DLLs), other files related to that code, and a descriptive manifest that includes information like the package's version number.

You can download the `IBMXMSDotnetClient` NuGet package, which contains the both the `amqmdnetstd.dll` library and the `amqmxmsstd.dll` library, from the NuGet Gallery, which is the central package repository used by all package authors and consumers.

Note: V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, the NuGet package contains libraries built using .NET 6 as the target framework.

Removed The XMS .NET client library built using .NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, IBM MQ supports .NET 8 applications using IBM MQ classes for XMS .NET. If you are using a .NET 6 application, you can run this application without any recompilation being required by making a small edit in the `runtimeconfig` file to set the `targetframeworkversion` to `"net8.0"`.

There are three ways of downloading the `IBMXMSDotnetClient` package:

- By using Microsoft Visual Studio. NuGet is distributed as a Microsoft Visual Studio extension. From Microsoft Visual Studio 2012, NuGet is pre-installed by default.
- From the command line using either the NuGet Package Manager or the .NET CLI.
- By using a web browser.

As for the redistributable package, you enable trace by using the environment variable **`XMS_TRACE_ON`**.

Procedure

- To download the `IBMXMSDotnetClient` package by using the Package Manager UI in Microsoft Visual Studio, complete the following steps:
  a) Right-click the .NET project and then click **Manage Nuget Packages**.

b) Click the **Browse** tab and search for `"IBMXMSDotnetClient"`.

c) Select the package and click **Install**.

During installation, the Package Manager provides progress information in the form of console statements.

- To download the `IBMXMSDotnetClient` package from the command line, choose one of the following options:
  - Using the NuGet Package Manager, enter the following command:

    ```
    Install-Package IBMXMSDotnetClient -Version 9.1.4.0
    ```

    During installation, the Package Manager provides progress information in the form of console statements. You can redirect the output to a log file.
  - Using the .NET CLI, enter the following command:

    ```
    dotnet add package IBMXMSDotnetClient --version 9.1.4
    ```
- Using a web browser, download the `IBMXMSDotnetClient` package from https://www.nuget.org/packages/IBMXMSDotnetClient.

Related concepts

“Installing IBM MQ classes for .NET” on page 537
IBM MQ classes for .NET, including samples, are installed with IBM MQ on Windows and Linux

IBM MQ Client for .NET license information

Related tasks

“Downloading IBM MQ classes for .NET from the NuGet repository” on page 542
The IBM MQ classes for .NET are available for downloading from the NuGet repository, so that they can be easily consumed by .NET Developers.

Setting up the messaging server environment

The topics in this section describe how to set up the messaging server environment to allow XMS applications to connect to a server.

About this task

For applications that connect to an IBM MQ queue manager, the IBM MQ client (or queue manager for bindings mode) is required.

There are currently no prerequisites for applications that use a real-time connection to a broker.

You must set up the messaging server environment before running any XMS applications, including the sample applications provided with XMS.

This section contains the following topics:

- “Configuring the queue manager and broker for an application that connects to an IBM MQ queue manager” on page 606
- “Installing IBM MQ classes for XMS .NET” on page 599
- “Configuring a broker for an application that uses a real-time connection to a broker” on page 607
- “Configuring the service integration bus for an application that connects to WebSphere Application Server” on page 608

Message listeners in XMS .NET

A message listener is used to receive messages asynchronously. Unlike the `MessageConsumer.receive()` call, the message listener does not block the calling thread, instead it delivers messages to an application specified callback method, typically the `onMessage` method.

Message delivery starts once the `Connection.Start()` method is called. Message delivery can be stopped and resumed anytime using the `Connection.Stop()` and `Connection.Start()` methods respectively.

Once the `Connection.Start()` method is called after setting a message listener to at least one consumer in a session, that session becomes an asynchronous session. Once a session becomes asynchronous, it is not possible to call any XMS .NET synchronous methods., for example, `MessageProducer.Send()`. Doing so results in an exception with IBM MQ reason code MQRC_HCONN_ASYNC_ACTIVE (2500).

Synchronous calls in an asynchronous session

`Session.Close` is the only synchronous call that is allowed in an asynchronous session. Applications can also make synchronous calls (except `Session.Close`) using the message listener callback method,that is, the `onMessage` method.

Other than these two options, you must stop the connection using the `Connection.Stop()` method for an application to make any synchronous call. After the calls are made, you must resume the connection again using the `Connection.Start()` method. which restarts message delivery.

How many asynchronous message consumers can a session have?

A session can have multiple asynchronous message consumers. But at any time message a message is delivered to only one consumer. What this practically means is, when a second message arrives while XMS .NET has called the `onMessage()` method of a consumer to deliver the first message, the second message will not be delivered to a consumer in the session until the `onMessage()` method returns.

The second message is delivered to a consumer in the session only after the `onMessage()` method returns. This is because a session manages message delivery to consumers using only one thread. This means that only one message can be delivered at a time, and the consumer could be any one.

If an application requires concurrent message delivery, that is, all the consumers must receive messages at the same time, then application must create multiple sessions and each must have one asynchronous message consumer.

The following examples show this feature more clearly.

In the first example, there are multiple asynchronous message consumers in a session. A session S has three asynchronous message consumers: AMC1, AMC2 and AMC3 which receive messages from three different destinations Q1, Q2 and Q3.

As there is only one session S, there is only message delivery thread to deliver messages to consumers AMC1, AMC2, and AMC3. When the session is delivering message to AMC1, the other two consumers AMC2 and AMC3 wait, even if there are messages in Q2 and Q3 ready for delivery.

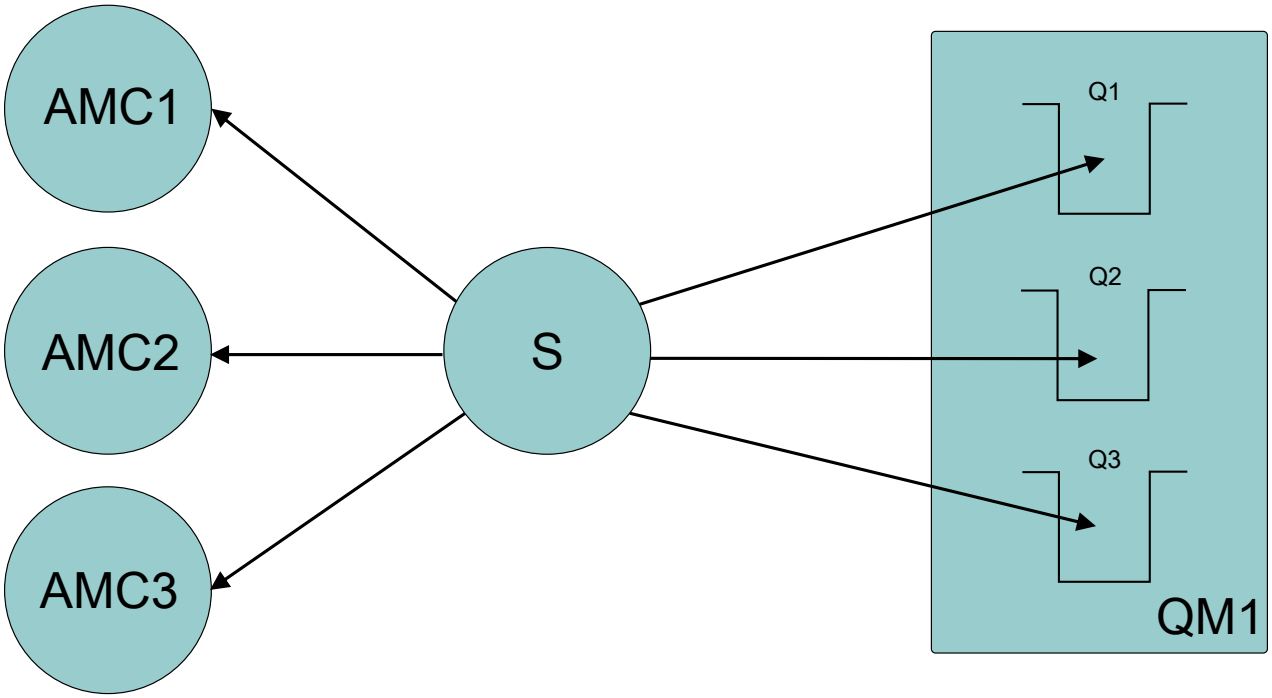


Figure 54. One session with three asynchronous message consumers

In the second case there are multiple sessions S1, S2, and S3, each having one asynchronous message consumer AMC1, AMC2, and AMC3 respectively. As there is one consumer for each session, messages are delivered to consumers concurrently.

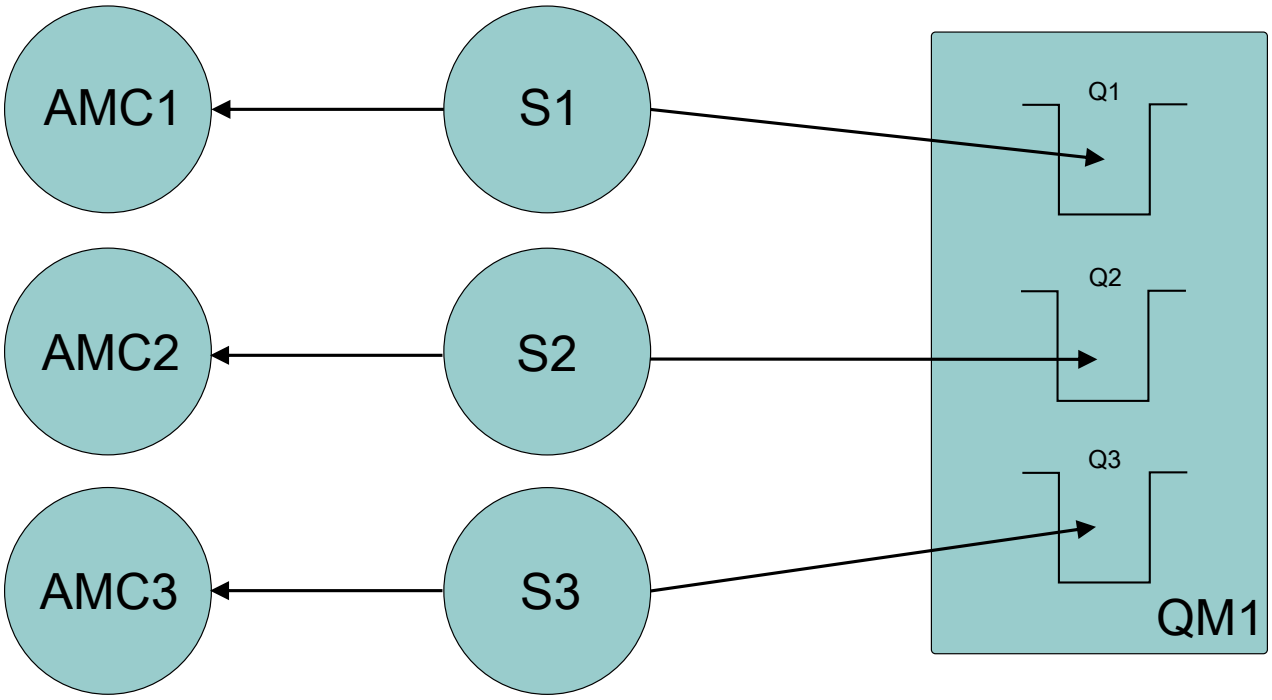


Figure 55. Multiple sessions, each with one asynchronous message consumer

This shows that if you require concurrent message delivery, you need multiple sessions.

Configuring the queue manager and broker for an application that connects to an IBM MQ queue manager

This section assumes that you are using IBM WebSphere MQ 7.0.1, or later. Before you can run an application that connects to an IBM MQ queue manager, you must configure the queue manager. For a publish/subscribe application, some additional configuration is required if you are using Queued publish/subscribe interface.

Before you begin

XMS works with IBM Integration Bus or WebSphere Message Broker 6.1 or later

Before starting this task, carry out the following steps:

- Make sure that your application has access to a queue manager that is running.
- If your application is a publish/subscribe application and uses Queued publish/subscribe interface, make sure that **PSMODE** attribute is set to ENABLED on the queue manager.
- Make sure that your application uses a connection factory whose properties are set appropriately to connect to the queue manager. If your application is a publish/subscribe application, make sure that the appropriate connection factory properties are set for using the broker. For more information about the properties of a connection factory, see Properties of ConnectionFactory.

About this task

You configure the queue manager and broker to run XMS applications in the same way that you configure the queue manager and queued publish/subscribe interface to run IBM MQ JMS applications. The following steps summarize what you need to do.

Procedure

1. On the queue manager, create the queues that your application needs.

   For an overview of how you create queues, see Defining queues.

   If your application is a publish/subscribe application and uses Queued publish/subscribe interface that needs access to IBM MQ classes for JMS system queues, wait until Step 4a before creating the queues.
2. Grant the user ID associated with your application the authority to connect to the queue manager, and the appropriate authority to access the queues.

   For an overview about authorization, see Securing. If your application connects to the queue manager in client mode, see also Clients and servers.
3. If your application connects to the queue manager in client mode, make sure that a server connection channel is defined at the queue manager and that a listener is started.

   You do not need to perform this step for each application that connects to the queue manager. One server connection channel definition and one listener can support all the applications that connect in client mode.
4. If your application is a publish/subscribe application, and uses Queued publish/subscribe interface, perform the following steps.
   a) On the queue manager, create the IBM MQ classes for JMS system queues by running the script of MQSC commands that are supplied with IBM MQ. Make sure that the user ID associated with the IBM Integration Bus or WebSphere Message Broker has the authority to access the queues.

      For information about where to find the script and how to run it, see Using IBM MQ classes for Java.

      Perform this step only once for the queue manager. The same set of IBM MQ classes for JMS system queues can support all XMS and IBM MQ classes for JMS applications that connect to the queue manager.

b) Grant the user ID associated with your application the authority to access the IBM MQ classes for JMS system queues.

   For information about what authorities the user ID needs, see Using IBM MQ classes for JMS.

c) For a broker of IBM Integration Bus or WebSphere Message Broker, create and deploy a message flow to service the queue where applications send messages that they publish.

   The basic message flow comprises an MQInput message processing node to read the published messages and a Publication message processing node to publish the messages.

   For information about how to create and deploy a message flow, see the IBM Integration Bus or WebSphere Message Broker product documentation available from the IBM Integration Bus product documentation library web page.

   You do not need to perform this step if a suitable message flow is already deployed at the broker.

Results

You can now start your application.

Configuring a broker for an application that uses a real-time connection to a broker

Before you can run an application that uses a real-time connection to a broker, you must configure that broker.

Before you begin

Before starting this task, you perform the following steps:

- Make sure that your application has access to a broker that is running.
- Make sure that your application uses a connection factory whose properties are set appropriately for a real-time connection to a broker. For more information about the properties of a connection factory, see Properties of ConnectionFactory.

About this task

You configure a broker to run XMS applications in the same way that you configure a broker to run IBM MQ classes for JMS applications. The following steps summarize what you need to do:

Procedure

1. Create and deploy a message flow to read messages from the TCP/IP port on which a broker is listening and publish the messages.

   You can do this in either of the following ways:

   - Create a message flow that contains a **Real-timeOptimizedFlow** message processing node.
   - Create a message flow that contains a **Real-timeInput** message processing node and a Publication message processing node.

   You must configure the **Real-timeOptimizedFlow** or **Real-timeInput** node to listen on the port used for real-time connections. In XMS, the default port number for real-time connections is 1506.

   You do not need to perform this step if a suitable message flow is already deployed at the broker.
2. If you require messages to be delivered to your application using IBM MQ classes for JMS, configure the broker to enable multicast. Configure the topics that must be multicast enabled, specifying a reliable quality of service for those topics requiring reliable multicast.
3. If your application supplies a user ID and a password when it connects to the broker, and you want the broker to authenticate your application using this information, configure the user name server and the broker for simple telnet-like password authentication.

Results

You can now start your application.

Configuring the service integration bus for an application that connects to WebSphere Application Server

Before you can run an application that connects to a WebSphere Application Server service integration technologies service integration bus, you must configure the service integration in the same way that you configure the service integration bus to run JMS applications that use the default messaging provider.

Before you begin

Before starting this task, you must do the following steps:

- Make sure that a messaging bus is created and that your server is added to the bus as a bus member.
- Make sure that your application has access to a service integration bus that contains at least one messaging engine that is running.
- If HTTP operation, is required then an HTTP messaging engine inbound transport channel must be defined. By default, channels for SSL and TCP are defined during the server installation.
- Make sure that your application uses a connection factory whose properties are set appropriately to connect to the service integration bus using a bootstrap server. The minimum information required is:
  - The provider endpoint, which describes the location and protocol to use when negotiating a connection to the messaging server (that is, via the bootstrap server). In its simplest form, for a server installed with default settings, the provide endpoint can be set to the host name of the server.
  - The name of the bus through which messages are sent.

  For more information about the properties of a connection factory, see Properties of ConnectionFactory.

About this task

Any queue or topic spaces that you require must be defined. By default a topic space called Default.Topic.Space is defined during the server installation but, if you require further topic spaces, you must create these topic spaces yourself. You do not need to predefine individual topics within a topic space, since the server instantiates these individual topics dynamically as required.

The following steps summarize what you need to do.

Procedure

1. Create the queues that your application needs for point-to-point messaging.
2. Create any additional topic spaces that your application needs for publish/subscribe messaging.

Results

You can now start your application.

Using the XMS sample applications

The XMS .NET sample applications provide an overview of the common features of each API. You can use them to verify your installation and messaging server setup and to help you build your own applications.

About this task

If you need help to create your own applications, you can use the sample applications as a starting point. Both the source and a compiled version are provided for each application. Review the sample source code and identify the key steps to create each required object for your application (ConnectionFactory, Connection, Session, Destination, and a Producer, or a Consumer, or both), and to set any specific properties that are needed to specify how you want your application to work. For more information,

see “Writing XMS .NET applications” on page 611. The samples are subject to change in future releases of XMS.

The following table shows the sets of sample applications (one for each API) that are supplied with XMS.

Table 81. Sample applications for XMS .NET

Name of sample	Description
SampleConsumerCS	A message consumer application that takes messages from a queue or subscribes to a topic.
SampleProducerCS	A message producer application that produces messages to a queue or on a topic.
SampleConfigCS	A configuration application that you can use to create an administered object repository that is file-based. The application contains a connection factory and destination for your particular connection settings. This administered object repository can then be used with each of the sample consumer and producer applications.

The samples that support the same functions in the various APIs have syntactical differences.

- The sample message consumer and producer applications both support the following functions:
  - Connections to IBM MQ, IBM Integration Bus (by using a real-time connection to a broker), and a WebSphere Application Server service integration bus
  - Administered object repository lookups by using the initial context interface
  - Connections to queues (IBM MQ and WebSphere Application Server service integration bus) and topics (IBM MQ, real-time connection to a broker, and WebSphere Application Server service integration bus)
  - Base, byte, map, object, stream, and text messages
- The sample message consumer application supports synchronous and asynchronous receive modes, and SQL Selector statements.
- The sample message producer application supports persistent and non-persistent delivery modes.

The samples can operate in one of two modes:

Simple mode
You can run the samples with the minimum user input.

Advanced mode
You can customize more finely the way in which the samples operate.

All the samples are compatible and can therefore operate across languages.

Windows IBM MQ supports .NET Core for XMS .NET applications in Windows environments. IBM MQ classes for .NET Standard, including samples, are installed by default as part of the standard IBM MQ installation.

Linux IBM MQ also supports .NET Core for applications in Linux environments.

The sample applications for XMS .NET are installed in `&MQINSTALL_PATH&/samp/dotnet/samples/cs/core/xms`.

For more information, see “Installing IBM MQ classes for XMS .NET” on page 599.

Running the .NET sample applications

You can run the .NET sample applications interactively in either simple or advanced mode, or noninteractively by using auto-generated or customized response files.

Before you begin

Before running any of the supplied sample applications, you must first set up the messaging server environment so that the applications can connect to a server. See "Setting up the messaging server environment" on page 603.

Procedure

To run a .NET sample application, complete the following steps:

Tip: When you are running a sample application, type ? at any time to get help about what to do next.

1. Select the mode in which you want to run the sample application.

   Type either `Advanced` or `Simple`.
2. Answer the questions.

   To select the default value, which is shown in the brackets at the end of the question, press Enter. To select a different value, type the appropriate value, and press Enter.

   Here is an example question:

   ```
   Enter connection type [wpm]:
   ```

   In this case, the default value is `wpm` (connection to a WebSphere Application Server service integration bus).

Results

When you run the sample applications, response files are generated automatically in the current working directory. Response file names are in the format *`connection_type-sample_type`*`.rsp`; for example, `wpm-producer.rsp`. If required, you can use the generated response file to rerun the sample application with the same options, so that you do not have to enter the options again.

Related tasks

Building the .NET sample applications
When you build a sample .NET application, an executable version of your chosen sample is created.

Building your own applications
You build your own applications like you build the sample applications.

Building the .NET sample applications

When you build a sample .NET application, an executable version of your chosen sample is created.

Before you begin

Install the appropriate compiler. This task assumes that you have Microsoft Visual Studio 2012 installed, and that you are familiar with using it.

Procedure

To build a .NET sample application, complete the following steps:

1. Click the `Samples.sln` solution file provided with the .NET samples.
2. Right-click the solution `Samples` in the Solution Explorer window and select **Build Solution**.

Results

An executable program is created in the appropriate subfolder of the sample, either `bin/Debug` or `bin/Release`, depending on the configuration that you have chosen. This program has the same name as the folder, with a suffix of CS. For example, if you are building the C# version of the message producer sample application, `SampleProducerCS.exe` is created in the `SampleProducer` folder.

Related tasks

Running the .NET sample applications

You can run the .NET sample applications interactively in either simple or advanced mode, or noninteractively by using auto-generated or customized response files.

Building your own applications

You build your own applications like you build the sample applications.

"Building your own applications" on page 611

You build your own applications like you build the sample applications.

Building your own applications

You build your own applications like you build the sample applications.

Before you begin

Install the appropriate compiler. This task assumes that you have Microsoft Visual Studio 2012 installed, and that you are familiar with using it.

Procedure

- Build your .NET application, as described in "Building the .NET sample applications" on page 610.

  For additional guidance on how to build your own applications, use the makefiles provided for each sample application.

  Tip: To assist with problem diagnosis in the event of a failure, you might find it helpful to compile applications with symbols included.

Related tasks

Running the .NET sample applications

You can run the .NET sample applications interactively in either simple or advanced mode, or noninteractively by using auto-generated or customized response files.

Building the .NET sample applications

When you build a sample .NET application, an executable version of your chosen sample is created.

Writing XMS .NET applications

This section provides information to help you when writing XMS .NET applications, including information about properties, data types, and error handling.

Before you begin

Deprecated V 9.4.0 V 9.4.0 From IBM MQ 9.4.0, in IBM MQ classes for XMS .NET, the methods WriteObject(), ReadObject(), CreateObjectMessage(), and the classes ObjectMessage and XmsObjectMessageImpl used for serialization and deserialization of data are deprecated.

Removed V 9.4.0 V 9.4.0 The XMS .NET client library built using.NET Standard 2.0, which was deprecated at IBM MQ 9.3.1, has been removed from the product at IBM MQ 9.4.0.

From IBM MQ 9.2.0, the number of XMS .NET dynamic link libraries has been significantly reduced, to a total of five. The five dynamic link libraries are:

- `IBM.XMS.dll` - includes all the national language messages
- `IBM.XMS.Comms.RMM.dll`

- Three policy dynamic link libraries:
  - `policy.8.0.IBM.XMS.dll`
  - `policy.9.0.IBM.XMS.dll`
  - `policy.9.1.IBM.XMS.dll`

In XMS .NET, all strings are passed using the native .NET string. Because this has a fixed encoding, no further information is required to interpret it. Therefore the XMSC_CLIENT_CCSID property is not required for XMS .NET applications.

About this task

This section contains the following topics:

Managed and unmanaged operations in .NET

Managed code is executed exclusively within the .NET common language runtime environment and is wholly dependent on the services provided by that runtime. An application is classed as unmanaged if any part of the application runs or calls services outside of the .NET common language runtime environment.

Certain advanced functionality cannot currently be supported within the managed .NET environment.

If your application requires some functionality that is not currently supported in the fully managed environment, then you can change your application to use the unmanaged environment without requiring substantial change to your application. However, you should note that the XMS stack makes use of unmanaged code when this selection is made.

Connections to an IBM MQ queue manager

Managed connections to WMQ_CM_CLIENT will not support non-TCP communications, and channel compression. However, these connections might be supported by using an unmanaged connection (WMQ_CM_CLIENT_UNMANAGED). For more information, see "Developing .NET applications" on page 535.

If you create a connection factory from an administered object in an unmanaged environment, you must manually change the value for the connection mode to XMSC_WMQ_CM_CLIENT_UNMANAGED.

Connections to a WebSphere Application Server service integration bus messaging engine

Connections to a WebSphere Application Server service integration bus messaging engine that require the use of the SSL protocol (including HTTPS) are not currently supported as managed code.

Windows Using the IBM MQ XMS .NET project template

The IBM MQ XMS .NET client offers you the ability to use a project template to assist you in developing your XMS .NET Core applications.

Before you begin

You must have Microsoft Visual Studio 2017, or later, and .NET Core 2.1 on your system.

You must copy the XMS .NET template from the

&MQ_INSTALL_ROOT&\tools\dotnet\samples\cs\core\xms\ProjectTemplates\IBMXMS.NETClientApp.zip

directory to the

&USER_HOME_DIRECTORY&\Documents\&Visual_Studio_Version&\Templates\ProjectTemplates

directory, where:

- *&MQ_INSTALL_ROOT* is the root directory of your installation
- *&USER_HOME_DIRECTORY* is your home directory.

You must stop and restart Microsoft Visual Studio to pick up the template.

About this task

The XMS .NET project template includes some common code that you can use to help develop your applications. With the in-built code, you can connect to the IBM MQ queue manager, and perform a put or a get operation by simply modifying the properties in the in-built code.

Procedure

1. Open Microsoft Visual Studio.
2. Click on **File**, followed by **New** and then **Project**.
3. In the *Create a new project window*, select `IBM XMS .NET Client App (.NET Core)` and click **Next**.
4. In the *Configure your new project* window, change the *Project name* of your project if you want to, and click **Create** to create the XMS .NET project.

   `XMSDotnetApp.cs` is the file that is created along with the project file. This file contains the code which connects to the queue manager, and performs a send and receive operation.

   The connection properties are set to default values:

   - WMQ_CONNECTION_NAME_LIST is set to *localhost(1414)*
   - XMSC.WMQ_CHANNEL is set to *DOTNET.SVRCONN*

   The queue is set to *Q1*, and you can modify these properties accordingly.
5. Compile and run the application.

Related concepts

IBM MQ components and features

.NET application runtime - Windows only

The threading model

General rules govern how a multithreaded application can use XMS objects.

- Only objects of the following types can be used concurrently on different threads:
  - ConnectionFactory
  - Connection
  - ConnectionMetaData
  - Destination
- A Session object can be used on only a single thread at any one time.

Exceptions to these rules are indicated by entries labeled "Thread context" in the interface definitions of the methods in IBM Message Service Client for .NET reference.

Properties in XMS .NET

A .NET application uses the methods in the PropertyContext interface to get and set the properties of objects. The handling of non-existent properties in the XMS .NET is broadly consistent with the JMS specification, and also with the C and C++ implementations of XMS.

XMS .NET properties and their values

The PropertyContext interface encapsulates methods that get and set properties. These methods are inherited, directly or indirectly, by the following classes:

- BytesMessage
- Connection
- ConnectionFactory
- ConnectionMetaData
- Destination
- MapMessage
- Message
- MessageConsumer
- MessageProducer
- ObjectMessage
- QueueBrowser
- Session
- StreamMessage
- TextMessage

If an application sets the value of a property, the new value replaces any previous value the property had. For more information about XMS properties, see Properties of XMS objects.

For ease of use, XMS property names and values in XMS are predefined as public constants in a struct called XMSC. The names of these constants are in the form XMSC.*constant*; for example, XMSC.USERID (a property name constant) and XMSC.DELIVERY_AS_APP (a value constant).

Additionally, you can access IBM MQ constants by using the IBM.XMS.MQC struct. If the IBM.XMS namespace is already imported, you can access the values for these properties in the form MQC.*constant*. For example, MQC.MQRO_COA_WITH_FULL_DATA.

If you have a hybrid application that uses both XMS .NET and IBM MQ classes for .NET and that imports both IBM.XMS and IBM.WMQ namespaces, then you must fully qualify the MQC struct namespace to ensure that each occurrence is unique.

Note: Some advanced functionality is not currently supported within the managed .NET environment. For more information, see “Managed and unmanaged operations in .NET” on page 612.

Handling of non-existent properties in XMS .NET

In JMS, accessing a non-existent property can result in a Java system exception when a method tries to convert the non-existent (null) value to the required type. If a property does not exist the following exceptions occur:

- getStringProperty and getObjectProperty return null
- getBooleanProperty returns false because Boolean.valueOf(null) returns false
- getIntProperty etc.throw java.lang.NumberFormatException because Integer.valueOf(null) throws the exception

If a property does not exist in XMS .NET, the following exceptions occur:

- GetStringProperty and GetObjectProperty (and GetBytesProperty) return null (which is the same as Java)
- GetBooleanProperty throws System.NullReferenceException
- GetIntProperty etc. throws System.NullReferenceException

This implementation is different from Java, but it is broadly consistent with the JMS specification, and with the XMS C and C++ interfaces. Like the Java implementation, XMS .NET propagates any exceptions from the System.Convert call to the caller. Unlike Java however, XMS explicitly throws NullReferenceExceptions rather than just using the native behavior of the .NET framework through passing null to system conversion routines. If your application sets a property to a String like "abc" and calls GetIntProperty, the System.FormatException thrown by Convert.ToInt32("abc") is propagated to the caller, which is consistent with Java. MessageFormatException is thrown only if the types used for setProperty and getProperty are incompatible. This behavior is also consistent with Java.

ConnectionFactories and Connection objects

A ConnectionFactory object provides a template that an application uses to create a Connection object. The application uses the Connection object to create a Session object.

For .NET, XMS application first uses an XMSFactoryFactory object to get a reference to a ConnectionFactory object that is appropriate to the required type of protocol. This ConnectionFactory object can then produce connections only for that protocol type.

An XMS application can create multiple connections, and a multithreaded application can use a single Connection object concurrently on multiple threads. A Connection object encapsulates a communications connection between an application and a messaging server.

A connection serves several purposes:

- When an application creates a connection, the application can be authenticated.
- An application can associate a unique client identifier with a connection. The client identifier is used to support durable subscriptions in the publish/subscribe domain. The client identifier can be set in two ways:

  The preferred way of assigning a connections client identifier, is to configure in a client-specific ConnectionFactory object using properties and transparently assign it to the connection it creates.

  An alternative way of assigning a client identifier is to use a provider-specific value that is set on the Connection object. This value does not override the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If an administratively specified identifier does exist, an attempt to override it with a provider-specific value causes an exception to be thrown. If an application explicitly sets an identifier, it must do it immediately after creating the connection and before any other action on the connection is taken; otherwise, an exception is thrown.

An XMS application typically creates a connection, one or more sessions, and a number of message producers and message consumers.

Creating a connection is relatively expensive in terms of system resources because it involves establishing a communications connection, and it might also involve authenticating the application.

Connection started and stopped mode

A connection can operate in either started or stopped mode.

When an application creates a connection, the connection is in stopped mode. When the connection is in stopped mode, the application can initialize sessions, and it can send messages but cannot receive them, either synchronously or asynchronously.

An application can start a connection by calling the `Start Connection` method. When the connection is in started mode, the application can send and receive messages. The application can then stop and restart the connection by calling the Stop Connection and `Start Connection` methods.

Connection closure

An application closes a connection by calling the Close Connection method. When an application closes a connection, XMS performs the following actions:

- It closes all the sessions associated with the connection and deletes certain objects associated with these sessions. For more information about which objects are deleted, see "Object deletion" on page 628. At the same time, XMS rolls back any transactions currently in progress within the sessions.
- It ends the communications connection with the messaging server.
- It releases the memory and other internal resources used by the connection.

XMS does not acknowledge the receipt of any messages that it has failed to acknowledge during a session, prior to closing the connection. For more information about acknowledging the receipt of messages, see "Message acknowledgment" on page 618.

Exception handling

XMS .NET exceptions are all derived from System.Exception. For more information, see "Error handling in XMS .NET" on page 633.

Connection to a service integration bus

An XMS application can connect to a WebSphere Application Server service integration bus either by using a direct TCP/IP connection or by using HTTP over TCP/IP.

The HTTP protocol can be used in situations where a direct TCP/IP connection is not possible. One common situation is when communicating through a firewall, such as when two enterprises exchange messages. Using HTTP to communicate through a firewall is often referred to as *HTTP tunneling*. HTTP tunneling, however, is inherently slower than using a direct TCP/IP connection because HTTP headers add significantly to the amount of data that is transferred, and because the HTTP protocol requires more communication flows than TCP/IP.

To create a TCP/IP connection, an application can use a connection factory whose XMSC_WPM_TARGET_TRANSPORT_CHAIN property is set to XMSC_WPM_TARGET_TRANSPORT_CHAIN_BASIC. This is the default value of the property. If the connection is created successfully, the XMSC_WPM_CONNECTION_PROTOCOL property of the connection is set to XMSC_WPM_CP_TCP.

To create a connection that uses HTTP, an application must use a connection factory whose XMSC_WPM_TARGET_TRANSPORT_CHAIN property is set to the name of an inbound transport chain, that is configured to use an HTTP transport channel. If the connection is created successfully, the XMSC_WPM_CONNECTION_PROTOCOL property of the connection is set to XMSC_WPM_CP_HTTP. For

information about how to configure transport chains, see Configuring transport chains in the WebSphere Application Server product documentation.

An application has a similar choice of communication protocols when connecting to a bootstrap server. The XMSC_WPM_PROVIDER_ENDPOINTS property of a connection factory is a sequence of one or more endpoint addresses of bootstrap servers. The bootstrap transport chain component of each endpoint address can be either XMSC_WPM_BOOTSTRAP_TCP, for a TCP/IP connection to a bootstrap server or XMSC_WPM_BOOTSTRAP_HTTP, for a connection that uses HTTP.

Sessions

A session is a single threaded context for sending and receiving messages.

An application can use a session to create messages, message producers, message consumers, queue browsers, and temporary destinations. An application can also use a session to run local transactions.

An application can create multiple sessions, where each session produces and consumes messages independently of the other sessions. If two message consumers in separate sessions (or even in the same session) subscribe to the same topic, each receives a copy of any message published on that topic.

Unlike a Connection object, a Session object cannot be used concurrently on different threads. Only the Close Session method of a Session object can be called from a thread other than the one that the Session object is using at the time. The Close Session method ends a session and releases any system resources allocated to the session.

If an application must process messages concurrently on more than one thread, the application must create a session on each thread, and then use that session for any send or receive operation within that thread.

Transacted sessions

XMS applications can run local transactions. A *local transaction* is a transaction that involves changes only to the resources of the queue manager or service integration bus to which the application is connected.

The information in this topic is relevant only if an application connects to an IBM MQ queue manager or a WebSphere Application Server service integration bus. The information is not relevant for a real-time connection to a broker.

To run local transactions, an application must first create a transacted session by calling the Create Session method of a Connection object, specifying as a parameter that the session is transacted. Subsequently, all messages sent and received within the session are grouped into a sequence of transactions. A transaction ends when the application commits or rolls back the messages it has sent and received since the transaction began.

To commit a transaction, an application calls the Commit method of the Session object. When a transaction is committed, all messages sent within the transaction become available for delivery to other applications, and all messages received within the transaction are acknowledged so that the messaging server does not attempt to deliver them to the application again. In the point-to-point domain, the messaging server also removes the received messages from their queues.

To roll back a transaction, an application calls the Rollback method of the Session object. When a transaction is rolled back, all messages sent within the transaction are discarded by the messaging server, and all messages received within the transaction become available for delivery again. In the point-to-point domain, the messages that were received are put back on their queues and become visible to other applications again.

A new transaction starts automatically when an application creates a transacted session or calls the Commit or Rollback method. Therefore, a transacted session always has an active transaction.

When an application closes a transacted session, an implicit rollback occurs. When an application closes a connection, an implicit rollback occurs for all the connection's transacted sessions.

A transaction is wholly contained within a transacted session. A transaction cannot span sessions. This means that it is not possible for an application to send and receive messages in two or more transacted sessions and then commit or roll back all these actions as a single transaction.

Related concepts

Message acknowledgment
Every session that is not transacted has an acknowledgment mode that determines how messages received by the application are acknowledged. Three acknowledgment modes are available, and the choice of acknowledgment mode affects the design of the application.

Message delivery
XMS supports persistent and nonpersistent modes of message delivery, and asynchronous and synchronous delivery of messages.

Managed IBM MQ XA transactions through XMS
Managed IBM MQ XA transactions can be used through XMS.

Message acknowledgment

Every session that is not transacted has an acknowledgment mode that determines how messages received by the application are acknowledged. Three acknowledgment modes are available, and the choice of acknowledgment mode affects the design of the application.

Note: This topic is relevant only if an application connects to an IBM MQ queue manager or a WebSphere Application Server service integration bus. The information is not relevant for a real-time connection to a broker.

XMS uses the same mechanism for acknowledging the receipt of messages that JMS uses.

If a session is not transacted, the way that messages received by the application are acknowledged is determined by the acknowledgment mode of the session. There are three acknowledgment modes:

XMSC_AUTO_ACKNOWLEDGE
The session automatically acknowledges each message received by the application.

If messages are delivered synchronously to the application, the session acknowledges receipt of a message every time a Receive call completes successfully. If the application receives a message successfully, but a failure prevents acknowledgment from occurring, the message becomes available for delivery again. The application must therefore be able to handle a message that is redelivered.

XMSC_DUPS_OK_ACKNOWLEDGE
The session acknowledges the messages received by the application at times it selects.

Using this acknowledgment mode reduces the amount of work the session must do, but a failure that prevents message acknowledgment might result in more than one message becoming available for delivery again. The application must therefore be able to handle messages that are redelivered.

XMSC_CLIENT_ACKNOWLEDGE
The application acknowledges the messages it receives by calling the Acknowledge method of the Message class.

The application can acknowledge the receipt of each message individually, or it can receive a batch of messages and call the Acknowledge method only for the last message it receives. When the Acknowledge method is called all messages received since the last time the method was called are acknowledged.

In conjunction with any of these acknowledgment modes, an application can stop and restart the delivery of messages in a session by calling the Recover method of the Session class. Messages whose receipt was previously unacknowledged are redelivered. However, they might not be delivered in the same sequence in which they were previously delivered. In the meantime, higher priority messages might have arrived, and some of the original messages might have expired. In the point-to-point domain, some of the original messages might have been consumed by another application.

An application can determine whether a message is being re-delivered by examining the contents of the JMSRedelivered header field of the message. The application does this by calling the Get JMSRedelivered method of the Message class.

Related concepts

Transacted sessions

XMS applications can run local transactions. A *local transaction* is a transaction that involves changes only to the resources of the queue manager or service integration bus to which the application is connected.

Message delivery
XMS supports persistent and nonpersistent modes of message delivery, and asynchronous and synchronous delivery of messages.

Managed IBM MQ XA transactions through XMS
Managed IBM MQ XA transactions can be used through XMS.

Message delivery

XMS supports persistent and nonpersistent modes of message delivery, and asynchronous and synchronous delivery of messages.

Message delivery modes

XMS supports two modes of message delivery:

- Persistent

  Persistent messages are delivered once. A messaging server takes special precautions, such as logging the messages, to ensure that persistent messages are not lost in transit, even in the event of a failure.
- Nonpersistent

  Nonpersistent messages are delivered no more than once. Nonpersistent messages are less reliable than persistent messages because they can be lost in transit in the event of a failure.

The choice of delivery mode is a trade-off between reliability and performance. Nonpersistent messages are typically transported more quickly than persistent messages.

Asynchronous message delivery

XMS uses one thread to handle all asynchronous message deliveries for a session. This means that only one message listener function or one `onMessage()` method can run at a time.

If more than one message consumer in a session is receiving messages asynchronously, and a message listener function or `onMessage()` method is delivering a message to a message consumer, then any other message consumers that are waiting for the same message must continue to wait. Other messages that are waiting to be delivered to the session must also continue to wait.

If an application requires concurrent delivery of messages, create more than one session so that XMS uses more than one thread to handle asynchronous message delivery. In this way, more than one message listener function or `onMessage()` method can run concurrently.

A session is not made asynchronous by assigning a message listener to a consumer. A session becomes asynchronous only when the `Connection.Start` method is called. All synchronous calls are permitted until the `Connection.Start` method is called. Message delivery to consumers start when the `Connection.Start` is called.

If synchronous calls, such as creation of a consumer or producer, must be made on an asynchronous session, the `Connection.Stop` must be called. A session can be resumed by calling the `Connection.Start` method to start delivery of messages. The only exception to this is the Session message delivery thread, which is the thread that delivers messages to the callback function. This thread can to make any call on session (except a Close call) in the message callback function.

Note: In Unmanaged mode, the MQDISC call within a call-back function is not supported by the IBM MQ .NET client. So, the client application cannot Create or Close sessions within the MessageListener callback in Asynchronous receive mode. Create and dispose the session outside of the `MessageListener` method.

Synchronous message delivery

Messages are delivered synchronously to an application if the application uses the Receive methods of MessageConsumer objects.

Using the Receive methods, an application can wait a specified period of time for a message, or it can wait indefinitely. Alternatively, if an application does not want to wait for a message, it can use the Receive with No Wait method.

Related concepts

Transacted sessions
XMS applications can run local transactions. A *local transaction* is a transaction that involves changes only to the resources of the queue manager or service integration bus to which the application is connected.

Message acknowledgment
Every session that is not transacted has an acknowledgment mode that determines how messages received by the application are acknowledged. Three acknowledgment modes are available, and the choice of acknowledgment mode affects the design of the application.

Managed IBM MQ XA transactions through XMS
Managed IBM MQ XA transactions can be used through XMS.

Managed IBM MQ XA transactions through XMS

Managed IBM MQ XA transactions can be used through XMS.

To use XA transactions through XMS, a transacted session has to be created. When XA transaction is in use, the transaction control is through Distributed Transaction Coordinator (DTC) global transactions and it is not though XMS sessions. When using XA transactions, `Session.commit` or `Session.rollback` cannot be issued on the XMS session. Instead, use the `Transscope.Commit` or `Transscope.Rollback` DTC methods commit or roll back the transactions. If a session is used for XA transaction, the producer or consumer that are created using the session must be a part of the XA transaction. They cannot be used for any operations outside the XA transaction scope. They cannot be used for operations like `Producer.send` or `Consumer.receive` outside the XA transaction.

An `IllegalStateException` exception object is thrown if:

- XA transacted session is used for `Session.commit` or `Session.rollback`.
- Producer or consumer objects which are once used in XA transacted session are used out side the XA transaction scope.

The XA transactions are not supported in asynchronous consumers.

Note:

1. A close might be issued on the `Producer`, `Consumer`, `Session`, or `Connection` object before the XA transaction commit. In which cases the messages in the transaction are rolled back. Similarly, if the connection is broken before the XA transaction commit, all the messages in the transaction are rolled back. For a `Producer` object, a rollback means that the messages are not put on the queue. For a `Consumer` object, a rollback means that the messages remain on the queue.
2. If a `Producer` object puts a message with `TimeToLive` in the `TransactionScope` and a `commit` is issued after the time is elapsed, the message can expire before the `commit` is issued. In this case, the message is not made available to `Consumer` objects.
3. `Session` objects are not supported across threads. The use of transactions with `Session` objects that are shared across threads is not supported.

Related concepts

Transacted sessions
XMS applications can run local transactions. A *local transaction* is a transaction that involves changes only to the resources of the queue manager or service integration bus to which the application is connected.

Message acknowledgment

Every session that is not transacted has an acknowledgment mode that determines how messages received by the application are acknowledged. Three acknowledgment modes are available, and the choice of acknowledgment mode affects the design of the application.

Message delivery
XMS supports persistent and nonpersistent modes of message delivery, and asynchronous and synchronous delivery of messages.

Destinations

An XMS application uses a Destination object to specify the destination of messages that are being sent, and the source of messages that are being received.

An XMS application can either create a Destination object at run time, or obtain a predefined destination from the repository of administered objects.

As with a ConnectionFactory, the most flexible way for an XMS application to specify a destination is to define it as an administered object. Using this approach, applications written in C, C++, and .NET languages, and Java, can share definitions of the destination. The properties of administered Destination objects can be changed without changing any code.

Destinations in .NET

In .NET, destinations are created according to protocol type and can be used only on the protocol type for which they are created. Two methods are provided for creating destinations, one for topics and one for queues:

- `IDestination CreateTopic(String topic);`
- `IDestination CreateQueue(String queue);`

These methods are available on the following two objects in the .NET API:

- ISession
- XMSFactoryFactory

In both cases, these methods can accept a URI style string, which can include parameters, in the following format:

```
"topic://some/topic/name?priority=5"
```

Alternatively, these methods can accept a destination name only, that is, a name without a topic:// or queue:// prefix and without parameters. This means that the following URI style string:

```
CreateTopic("topic://some/topic/name");
```

would produce the same result as the following destination name:

```
CreateTopic("some/topic/name");
```

For more information, see IDestination.

As for WebSphere Application Server service integration bus JMS, topics can also be specified in a shorthand form, which includes both the *topicname* and *topicspace* but cannot include parameters:

```
CreateTopic("topicspace:topicname");
```

Topic uniform resource identifiers

The topic uniform resource identifier (URI) specifies the name of the topic; it can also specify one or more properties for it.

The URI for a topic begins with the sequence topic://, followed by the name of the topic and (optional) a list of name-value pairs that set the remaining topic properties. A topic name cannot be empty.

Here is an example in a fragment of .NET code:

```
topic = session.CreateTopic("topic://Sport/Football/Results?multicast=7");
```

For more information about the properties of a topic, including the name and valid values that you can use in a URI, see Properties of Destination.

When specifying a topic URI for use in a subscription, wildcards can be used. The syntax for these wildcards depends on the connection type and broker version; the following option is available:

- WebSphere Application Server service integration bus

WebSphere Application Server service integration bus

WebSphere Application Server service integration bus uses the following wildcard characters:

* to match any characters at one level in the hierarchy
// to match 0 or more levels
//. to match 0 or more levels (at the end of a Topic expression)

Table 82 on page 622 gives some examples of how to use this wildcard scheme.

Table 82. Example URIs using wildcard scheme for WebSphere Application Server service integration bus

Uniform Resource Identifier	Matches	Examples
"topic://Sport/*ball/Results"	All topics with a single hierarchical level name ending in "ball" between Sport and Results	"topic://Sport/Football/Results" and "topic://Sport/Netball/Results"
"topic://Sport//Results"	All topics starting with "Sport/" and ending in "/Results"	"topic://Sport/Football/Results" and "topic://Sport/Hockey/National/Div3/Results"
"topic://Sport/Football//."	All topics starting with "Sport/Football/"	"topic://Sport/Football/Results" and "topic://Sport/Football/TeamNews/Signings/Managerial"
"topic://Sport/*ball//Results//."	Topics	"topic://Sport/Football/Results" and "topic://Sport/Netball/National/Div3/Results/2002/November"

Related concepts

Queue uniform resource identifiers
The URI for a queue specifies the name of the queue; it can also specify one or more properties of the queue.

Temporary destinations

XMS applications can create and use temporary destinations.

Queue uniform resource identifiers

The URI for a queue specifies the name of the queue; it can also specify one or more properties of the queue.

The URI for a queue begins with the sequence `queue://`, followed by the name of the queue; it might also include a list of name-value pairs that set the remaining queue properties.

For IBM MQ queues (but not for WebSphere Application Server default messaging provider queues), the queue manager on which the queue resides may be specified before the queue, with a / separating the queue manager name from the queue name.

If a queue manager is specified, then it must be the one to which XMS is directly connected for the connection using this queue, or it must be accessible from this queue. Remote queue managers are only supported for retrieving messages from queues, not for putting messages onto queues. For full details, refer to the IBM MQ queue manager documentation.

If no queue manager is specified, then the extra / separator is optional, and its presence or absence makes no difference to the definition of the queue.

The following queue definitions are all equivalent for an IBM MQ queue called QB on a queue manager called QM_A, to which XMS is directly connected:

```
queue://QB
queue:///QB
queue://QM_A/QB
```

Related concepts

Topic uniform resource identifiers
The topic uniform resource identifier (URI) specifies the name of the topic; it can also specify one or more properties for it.

Temporary destinations
XMS applications can create and use temporary destinations.

Temporary destinations

XMS applications can create and use temporary destinations.

An application typically uses a temporary destination to receive replies to request messages. To specify the destination where a reply to a request message is to be sent, an application calls the Set JMSReplyTo method of the Message object representing the request message. The destination specified on the call can be a temporary destination.

Although a session is used to create a temporary destination, the scope of a temporary destination is actually the connection that was used to create the session. Any of the connection's sessions can create message producers and message consumers for the temporary destination. The temporary destination remains until it is explicitly deleted, or the connection ends, whichever happens first.

When an application creates a temporary queue, a queue is created in the messaging server to which the application is connected. If the application is connected to a queue manager, a dynamic queue is created from the model queue whose name is specified by the XMSC_WMQ_TEMPORARY_MODEL property, and the prefix that is used to form the name of the dynamic queue is specified by the XMSC_WMQ_TEMP_Q_PREFIX property. If the application is connected to a service integration bus, a temporary queue is created in the bus, and the prefix that is used to form the name of the temporary queue is specified by the XMSC_WPM_TEMP_Q_PREFIX property.

When an application that is connected to a service integration bus creates a temporary topic, the prefix that is used to form the name of the temporary topic is specified by the XMSC_WPM_TEMP_TOPIC_PREFIX property.

Related concepts

Topic uniform resource identifiers

The topic uniform resource identifier (URI) specifies the name of the topic; it can also specify one or more properties for it.

Queue uniform resource identifiers
The URI for a queue specifies the name of the queue; it can also specify one or more properties of the queue.

Message producers

In XMS, a message producer can be created either with a valid destination or with no associated destination. When creating a message producer with a null destination, a valid destination needs to be specified when sending a message.

Message producers with associated destination

In this scenario, the message producer is created using a valid destination. During the send operation, the destination need not be specified.

Message producers with no associated destination

In XMS .NET, a message producer can be created with a null destination.

To create a message producer with no associated destination when using the .NET API, NULL must be passed as a parameter into the `CreateProducer()` method of the ISession object (for example, `session.CreateProducer(null)`). However a valid destination must be specified when the message is sent.

Message consumers

Message consumers can be classified as durable and non-durable subscribers and synchronous and asynchronous message consumers.

Durable subscribers

A durable subscriber is a message consumer that receives all messages published on a topic, including messages published while the subscriber is inactive.

Attention: This information is relevant only if an application connects to an IBM MQ queue manager or a WebSphere Application Server service integration bus. The information is not relevant for a real-time connection to a broker.

To create a durable subscriber for a topic, an application calls the Create Durable Subscriber method of a Session object, specifying as parameters a name that identifies the durable subscription and a Destination object representing the topic. The application can create a durable subscriber with or without a message selector, and it can specify whether the durable subscriber is to receive messages published by its own connection.

The session used to create a durable subscriber must have an associated client identifier. The client identifier is the same as that associated with the connection that is used to create the session; it is specified as described in “ConnectionFactories and Connection objects” on page 615.

The name that identifies the durable subscription must be unique within the client identifier, and therefore the client identifier forms part of the full, unique identifier of the durable subscription. The messaging server maintains a record of the durable subscription and ensures that all messages published on the topic are retained until they are acknowledged by the durable subscriber or they expire.

The messaging server continues to maintain the record of the durable subscription even after the durable subscriber closes. To reuse a durable subscription that was created previously, an application must create a durable subscriber specifying the same subscription name, and using a session with the same client identifier, as those associated with the durable subscription. Only one session at a time can have a durable subscriber for a particular durable subscription.

The scope of a durable subscription is the messaging server that is maintaining a record of the subscription. If two applications connected to different messaging servers each create a durable subscriber using the same subscription name and client identifier, two completely independent durable subscriptions are created.

To delete a durable subscription, an application calls the Unsubscribe method of a Session object, specifying as a parameter the name that identifies the durable subscription. The client identifier associated with the session must be the same as that associated with the durable subscription. The messaging server deletes the record of the durable subscription that it is maintaining and does not send any more messages to the durable subscriber.

To change an existing subscription, an application can create a durable subscriber using the same subscription name and client identifier, but specifying a different topic, or message selector (or both). Changing a durable subscription is equivalent to deleting the subscription and creating a new one.

For an application that connects to an IBM MQ queue manager, XMS manages the subscriber queues. Hence the application is not required to specify a subscriber queue. XMS will ignore the subscriber queue if specified.

Note that you cannot change the subscriber queue for a durable subscription. The only way to change the subscriber queue is to delete the subscription and create a new one.

For an application that connects to a service integration bus, each durable subscriber must have a designated durable subscription home. To specify the durable subscription home for all durable subscribers that use the same connection, set the XMSC_WPM_DUR_SUB_HOME property of the ConnectionFactory object that is used to create the connection. To specify the durable subscription home for an individual topic, set the XMSC_WPM_DUR_SUB_HOME property of the Destination object representing the topic. A durable subscription home must be specified for a connection before an application can create a durable subscriber that uses the connection. Any value specified for a destination overrides the value specified for the connection.

Synchronous message consumers

The synchronous message consumer receives the messages from a queue synchronously and receives one message at a time. When the `Receive(wait interval)` method is used; the call waits only a specified period of time in milliseconds for a message, or until the message consumer is closed.

If the ReceiveNoWait() method is used, the synchronous message consumer receives messages without any delay; if the next message is available, it is received immediately, otherwise a pointer to a null Message object is returned.

Asynchronous message consumers

The asynchronous message consumer receives message from a queue asynchronously. The message listener registered by the application is invoked whenever a new message is available on the queue.

Poison messages in XMS

A poison message is a message that cannot be processed by a receiving MDB application. If a poison message is encountered, the XMS MessageConsumer object can requeue it according to two queue properties, **BOQUEUE**, and **BOTHRESH**.

In some circumstances, a message delivered to an MDB might be rolled back onto an IBM MQ queue. This can happen, for example, if a message is delivered within a unit of work that is subsequently rolled back. A message that is rolled back is generally delivered again, but a badly formatted message might repeatedly cause an MDB to fail and therefore cannot be delivered. Such a message is called a poison message. You can configure IBM MQ so that the poison message is automatically transferred to another queue for further investigation or is discarded. For information about how to configure IBM MQ in this way, see “Handling poison messages in ASF” on page 627.

Sometimes, a badly formatted message arrives on a queue. In this context, badly formatted means that the receiving application cannot process the message correctly. Such a message can cause the receiving application to fail and to back out this badly formatted message. The message can then be repeatedly

delivered to the input queue and repeatedly backed out by the application. These messages are known as poison messages. The XMS MessageConsumer object detects poison messages and reroutes them to an alternative destination.

The IBM MQ queue manager keeps a record of the number of times that each message has backed out. When this number reaches a configurable threshold value, the message consumer requeues the message to a named backout queue. If this requeuing fails for any reason, the message is removed from the input queue and either requeued to the dead-letter queue, or discarded.

XMS ConnectionConsumer objects handle poison messages in the same way and using the same queue properties. If multiple connection consumers are monitoring the same queue, it is possible that the poison message may be delivered to an application more times than the threshold value before the requeue occurs. This behavior is due to the way individual connection consumers monitor queues and requeue poison messages.

The threshold value and the name of the back out queue are attributes of an IBM MQ queue. The names of the attributes are **BackoutThreshold** and **BackoutRequeueQName**. The queue they apply to is as follows:

- For point-to-point messaging, this is the underlying local queue. This is important when message consumers and connection consumers use queue aliases.
- For publish/subscribe messaging in IBM MQ messaging provider normal mode, it is the model queue from which the Topic's managed queue is created.
- For publish/subscribe messaging in IBM MQ messaging provider migration mode, it is the CCSUB queue defined on the TopicConnectionFactory object, or the CCDSUB queue defined on the Topic object.

To set the **BackoutThreshold** and **BackoutRequeueQName** attributes, issue the following MQSC command:

```
ALTER QLOCAL(your.queue.name) BOTHRESH(threshold value)
BOQUEUE(your.backout.queue.name)
```

For publish/subscribe messaging, if your system creates a dynamic queue for each subscription, these attribute values are obtained from the IBM MQ classes for JMS model queue, SYSTEM.JMS.MODEL.QUEUE. To alter these settings, use:

```
ALTER QMODEL(SYSTEM.JMS.MODEL.QUEUE) BOTHRESH(threshold value)
BOQUEUE(your.backout.queue.name)
```

If the backout threshold value is zero, poison message handling is disabled, and poison messages remain on the input queue. Otherwise, when the backout count reaches the threshold value, the message is sent to the named backout queue.

If the backout count reaches the threshold value, but the message cannot go to the backout queue, the message is sent to the dead-letter queue or, if the message is nonpersistent, it is discarded.

This situation occurs if the backout queue is not defined, or if the MessageConsumer object cannot send the message to the backout queue.

Configuring your system to perform poison message handling

The queue that XMS .NET uses when inquiring the **BOTHRESH** and **BOQNAME** attributes depends on the style of messaging being performed:

- For point-to-point messaging, this is the underlying local queue. This is important when an XMS .NET application is consuming messages from either alias queues or cluster queues.
- For publish/subscribe messaging, a managed queue is created to hold the messages for an application. XMS .NET queries the managed queue to determine the values for the **BOTHRESH** and **BOQNAME** attributes.

The managed queue is created from a model queue associated with the Topic object that the application has subscribed to, and inherits the values of the **BOTHRESH** and **BOQNAME** attributes from the model queue. The model queue that is used depends on whether the receiving application has taken out a durable or non-durable subscription:

- The model queue used for durable subscriptions is specified by the **MDURMDL** attribute of the Topic. The default value of this attribute is `SYSTEM.DURABLE.MODEL.QUEUE`.
- For non-durable subscriptions, the model queue that is used is specified by the **MNDURMDL** attribute. The default value of the **MNDURMDL** attribute is `SYSTEM.NDURABLE.MODEL.QUEUE`.

When inquiring the **BOTHRESH** and **BOQNAME** attributes, XMS .NET:

- Opens the local queue, or the target queue for an alias queue.
- Inquires the **BOTHRESH** and **BOQNAME** attributes.
- Closes the local queue, or the target queue for an alias queue.

The open options that are used when opening a local queue, or the target queue for an alias queue, depend on the version of IBM MQ being used:

- For IBM MQ 9.1.0 Fix Pack 4 Long Term Support and earlier, and IBM MQ 9.1.4 Continuous Delivery and earlier: If the local queue, or the target queue for an alias queue, is a cluster queue, then XMS .NET opens the queue with the `MQOO_INPUT_AS_Q_DEF`, `MQOO_INQUIRE` and `MQOO_FAIL_IF_QUIESCING` options. This means that the user running the receiving application must have inquire and get access to the local instance of the cluster queue.

  XMS .NET opens all other types of local queue with the open options `MQOO_INQUIRE` and `MQOO_FAIL_IF_QUIESCING`. In order for XMS .NET to query the values of the attributes, the user running the receiving application must have inquire access on the local queue.

To move poison messages to either a backout requeue queue or the queue manager's dead letter queue, you must grant the user running the application `put` and `passall` authorities.

Handling poison messages in ASF

When you use Application Server Facilities (ASF), the ConnectionConsumer, rather than the MessageConsumer, processes poison messages. The ConnectionConsumer re-queues messages according to the **BackoutThreshold** and **BackoutRequeueQName** properties of the queue.

When an application uses ConnectionConsumers, the circumstances in which a message is backed out depend on the session that the application server provides:

- When the session is non-transacted, with AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE, a message is backed out only after a system error, or if the application terminates unexpectedly.
- When the session is non-transacted with CLIENT_ACKNOWLEDGE, unacknowledged messages can be backed out by the application server calling `Session.recover()`.

  Typically, the client implementation of MessageListener or the application server calls `Message.acknowledge()`. `Message.acknowledge()` acknowledges all messages delivered on the session so far.
- When the session is transacted, unacknowledged messages can be backed out by the application server calling `Session.rollback()`.

Queue browsers

An application uses a queue browser to browse messages on a queue without removing them.

To create a queue browser, an application calls the Create Queue Browser method of an ISession object, specifying as a parameter a Destination object that identifies the queue to be browsed. The application can create a queue browser with or without a message selector.

After creating a queue browser, the application can call the GetEnumerator method of the IQueueBrowser object to get a list of the messages on the queue. The method returns an enumerator that encapsulates

a list of Message objects. The order of the Message objects in the list is the same as the order in which the messages would be retrieved from the queue. The application can then use the enumerator to browse each message in turn.

The enumerator is updated dynamically as messages are put on the queue and removed from the queue. Each time the application calls IEnumerator.MoveNext() to browse the next message on the queue, the message reflects the current contents of the queue.

An application can call the GetEnumerator method more than once for a given queue browser. Each call returns a new enumerator. The application can therefore use more than one enumerator to browse the messages on a queue and maintain multiple positions within the queue.

An application can use a queue browser to search for a suitable message to remove from a queue, and then use a message consumer with a message selector to remove the message. The message selector can select the message according to the value of the JMSMessageID header field. For information about this and other JMS message header fields, see “Header fields in an XMS message” on page 646.

Requestors

An application uses a requestor to send a request message and then to wait for and to receive the reply.

Many messaging applications are based on algorithms that send a request message and then wait for a reply. XMS provides a class called Requestor to help with the development of this style of application.

To create a requestor, an application calls the Create Requestor constructor of the Requestor class, specifying as parameters a Session object and a Destination object that identifies where request messages are to be sent. The session must not be transacted nor have an acknowledgment mode of XMSC_CLIENT_ACKNOWLEDGE. The constructor automatically creates a temporary queue or topic where reply messages are to be sent.

After creating a requestor, the application can call the Request method of the Requestor object to send a request message and then wait for, and receive, a reply from the application that receives the request message. The call waits until the reply is received or until the session ends, whichever occurs first. Only one reply is required by the requestor for each request message.

When the application closes the requestor, the temporary queue or topic is deleted. The associated session, however, does not close.

Object deletion

When an application deletes an XMS object that it created, XMS releases the internal resources that have been allocated to the object.

When an application creates an XMS object, XMS allocates memory and other internal resources to the object. XMS retains these internal resources until the application explicitly deletes the object by calling the object's close or delete method, at which point XMS releases the internal resources. If an application tries to delete an object that is already deleted, the call is ignored.

When an application deletes a Connection or Session object, XMS deletes certain associated objects automatically and releases their internal resources. These are objects that were created by the Connection or Session object and have no function independent from the object. These objects are shown in Table 83 on page 628.

Note: if an application closes a connection with dependent sessions, all objects dependent on those sessions are also deleted. Only a Connection or Session object can have dependent objects.

Table 83. Objects that are deleted automatically

Deleted object	Method	Dependent objects that are deleted automatically
Connection	Close Connection	ConnectionMetaData and Session objects
Session	Close Session	MessageConsumer, MessageProducer, QueueBrowser, and Requestor objects

Data types for XMS .NET

XMS .NET supports System.Boolean, System.Byte, System.SByte, System.Char, System.String, System.Single, System.Double, System.Decimal, System.Int16, System.Int32, System.Int64, System.UInt16, System.UInt32, System.UInt64, and System.Object. Data types for XMS .NET are different from data types for XMS C/C++. You can use this topic to identify the corresponding data types.

The following table shows the corresponding XMS .NET and XMS C/C++ data types and briefly describes them.

Table 84. Data types for XMS .NET and XMS C/C++

XMS .NET type	XMS C/C++ type	Description
System.SByte	xmsSBYTE xmsINT8	Signed 8-bit value
System.Byte	xmsBYTE xmsUINT8	Unsigned 8-bit value
System.Int16	xmsINT16 xmsSHORT	Signed 16-bit value
System.UInt16	xmsUINT16 xmsUSHORT	Unsigned 16-bit value
System.Int32	xmsINT32 xmsINT	Signed 32-bit value
System.UInt32	xmsUINT32 xmsUINT	Unsigned 32-bit value
System.Int64	xmsLONG xmsINT64	Signed 64-bit value
System.UInt64	xmsULONG xmsUINT64	Unsigned 64-bit value
System.Char	xmsCHAR16	Unsigned 16-bit character (Unicode for .NET)
System.Single	xmsFLOAT	IEEE 32-bit float
System.Double	xmsDOUBLE	IEEE 64-bit float
System.Boolean	xmsBOOL	A True/False value
Not applicable	xmsCHAR	Signed or Unsigned 8-bit value (signed or unsigned depends on platform)
System.Decimal	Not applicable	96-bit signed integer times 10^0 through 10^{28}
System.Object	Not applicable	Base of all types
System.String	Not applicable	String type

XMS primitive types

XMS provides equivalents of the eight Java primitive types (byte, short, int, long, float, double, char, and boolean). This allows the interchange of messages between XMS and JMS without data becoming lost or corrupted.

Table 85 on page 630 lists the Java equivalent data type, size, and minimum and maximum value of each XMS primitive type.

Table 85. XMS data types and their Java equivalents

XMS data type	Compatible Java data type	Size	Minimum value	Maximum value
System.Boolean	boolean	32 bits	false	true
System.SBYTE	byte	8 bits	-2^7 (-128)	2^7-1 (127)
System.BYTE	byte	8 bits	-2^7 (-128)	2^7-1 (127)
System.CHAR	byte	8 bits	-2^7 (-128)	2^7-1 (127)
System.Int16	short	16 bits	-2^{15} (-32768)	2^{15}-1 (32767)
System.Int32	int	32 bits	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
System.Int64	long	64 bits	-2^{63} (-9223372036854775808)	2^{63}-1 (9223372036854775807)
System.Single	float	32 bits	-3.402823E-38 (to 7-digits precision)	3.402823E+38 (to 7-digits precision)
System.Double	double	64 bits	-1.79769313486231E-308 (to 15-digits precision)	1.79769313486231E+308 (to 15-digits precision)

Implicit conversion of a property value from one data type to another

When an application gets the value of a property, the value can be converted by XMS into another data type. Many rules govern which conversions are supported and how XMS performs the conversions.

A property of an object has a name and a value; the value has an associated data type, where the value of a property is also referred to as the *property type*.

An application uses the methods of the PropertyContext class to get and set the properties of objects. In order to get the value of a property, an application calls the method that is appropriate for the property type. For example, to get the value of an integer property, an application typically calls the GetIntProperty method.

However, when an application gets the value of a property, the value can be converted by XMS into another data type. For example, to get the value of an integer property, an application can call the GetStringProperty method, which returns the value of the property as a string. The conversions supported by XMS are shown in Table 86 on page 630.

Table 86. Supported conversions from a property type to other data types

Property type	Supported target data types
System.String	System.Boolean, System.Double, System.Float, System.Int32, System.Int64, System.SByte, System.Int16
System.Boolean	System.String, System.SByte, System.Int32, System.Int64, System.Int16
System.Char	System.String
System.Double	System.String

Table 86. Supported conversions from a property type to other data types (continued)	
Property type	**Supported target data types**
System.Float	System.String, System.Double
System.Int32	System.String, System.Int64
System.Int64	System.String
System.SByte	System.String, System.Int32, System.Int64, System.Int16
System.SByte array	System.String
System.Int16	System.String, System.Int32, System.Int64

The following general rules govern the supported conversions:

- Numeric property values can be converted from one data type to another provided no data is lost during the conversion. For example, the value of a property with data type System.Int32 can be converted into a value with data type System.Int64, but it cannot be converted into a value with data type System.Int16.
- A property value of any data type can be converted into a string.
- A string property value can be converted to any other data type provided the string is formatted correctly for the conversion. If an application attempts to convert a string property value that is not formatted correctly, XMS may return errors.
- If an application attempts a conversion that is not supported, XMS may return an error.

The following rules apply when a property value is converted from one data type to another:

- When converting a boolean property value to a string, the value true is converted to the string "true", and the value false is converted to the string "false".
- When converting a boolean property value to a numeric data type, including System.SByte, the value true is converted to 1, and the value false is converted to 0.
- When converting a string property value to a boolean value, the string "true" (not case-sensitive) or "1" is converted to true, and the string "false" (not case-sensitive) or "0" is converted to false. All other strings cannot be converted.
- When converting a string property value to a value with data type System.Int32, System.Int64, System.SByte, or System.Int16, the string must have the following format:

  [*blanks*][*sign*]*digits*

  The string components are defined as follows:

  blanks
  Optional leading blank characters.

  sign
  An optional plus sign (+) or minus sign (-) character.

  digits
  A contiguous sequence of digit characters (0-9). At least one digit character must be present.

  After the sequence of digit characters, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal integer.

  XMS may return an error if the string is not formatted correctly.
- When converting a string property value to a value with data type System.Double or System.Float, the string must have the following format:

  [*blanks*][*sign*][*digits*][*point*[*d_digits*]][*e_char*[*e_sign*]*e_digits*]

  The string components are defined as follows:

blanks
(Optional) Leading blank characters.

sign
(Optional) Plus sign (+) or minus sign (-) character.

digits
A contiguous sequence of digit characters (0-9). At least one digit character must be present in either *digits* or *d_digits*.

point
(Optional) Decimal point (.).

d_digits
A contiguous sequence of digit characters (0-9). At least one digit character must be present in either *digits* or *d_digits*.

e_char
An exponent character, which is either *E* or *e*.

e_sign
(Optional) Plus sign (+) or minus sign (-) character for the exponent.

e_digits
A contiguous sequence of digit characters (0-9) for the exponent. At least one digit character must be present if the string contains an exponent character.

After the sequence of digit characters, or the optional characters representing an exponent, the string can contain other characters that are not digit characters, but the conversion stops as soon as the first of these characters is reached. The string is assumed to represent a decimal floating point number with an exponent that is a power of 10.

XMS may return an error if the string is not formatted correctly.

- When converting a numeric property value to a string, including a property value with data type System.SByte, the value is converted to the string representation of the value as a decimal number, not the string containing the ASCII character for that value. For example, the integer 65 is converted to the string "65", not the string "A".
- When converting a byte array property value to a string, each byte is converted to the 2 hexadecimal characters that represent the byte. For example, the byte array {0xF1, 0x12, 0x00, 0xFF} is converted to the string "F11200FF".

Conversions from a property type to other data types are supported by the methods of both the Property and the PropertyContext classes.

Iterators

An iterator encapsulates a list of objects and a cursor that maintains the current position in the list. The concept of an Iterator, as available in IBM MQ Message Service Client (XMS) for C/C++, is implemented by using IEnumerator interface in IBM MQ Message Service Client (XMS) for .NET.

When an iterator is created, the position of the cursor is before the first object. An application uses an iterator to retrieve each object in turn.

The Iterator class of IBM MQ Message Service Client (XMS) for C/C++ is equivalent to the Enumerator class in Java. IBM MQ Message Service Client (XMS) for .NET is similar to Java and uses an IEnumerator interface.

An application can use an IEnumerator to perform the following tasks:

- To get the properties of a message
- To get the name-value pairs in the body of a map message
- To browse the messages on a queue
- To get the names of the JMS defined message properties supported by a connection

Error handling in XMS .NET

XMS .NET exceptions are all derived from System.Exception.

XMS .NET exceptions

XMS method calls can throw specific XMS exceptions such as MessageFormatException, general XMSExceptions, or system exceptions such as NullReferenceException.

Write applications to catch any of these errors, either in specific catch blocks or in general `System.Exception` catch blocks, as appropriate to the your application's requirements.

XMS error and exception codes

XMS uses a range of error codes to indicate failures. These error codes are not explicitly listed in this documentation because they may vary from release to release. Only XMS exception codes (in the format XMS_X_...) are documented because they remain the same across releases of XMS.

Related information

MQException.NET class
Common SSL error codes thrown by XMS .NET client libraries
FFDC configuration for XMS .NET applications
Tracing XMS .NET applications

Using message and exception listeners in .NET

A .NET application uses a message listener to receive messages asynchronously, and it uses an exception listener to be notified asynchronously of a problem with a connection.

About this task

The functionality of both the message and exception listeners is the same for .NET and for C++. However, there are some small implementation differences.

Procedure

- To set up a message listener in order to receive messages asynchronously, complete the following steps:
  a) Define a method that matches the signature of the message listener delegate.
  The method that you define can be either a static or an instance method and can be defined in any accessible class. The delegate signature is as follows:

```
public delegate void MessageListener(IMessage msg);
```

  and so you could define the method as:

```
void SomeMethodName(IMessage msg);
```

  b) Instantiate this method as a delegate using something similar to the following example:

```
MessageListener OnMsgMethod = new MessageListener(SomeMethodName)
```

  c) Register the delegate with one or more consumers by setting it to the MessageListener property of the consumer:

```
consumer.MessageListener = OnMsgMethod;
```

You can remove the delegate by setting the MessageListener back to null:

```
consumer.MessageListener = null;
```

- To set up an exception listener, complete the following steps.

  The exception listener works in much the same way as the message listener, but has a different delegate definition and is assigned to the connection rather then the message consumer. This is the same as for C++.

  a) Define the method.

  The delegate signature is as follows:

```
public delegate void ExceptionListener(Exception ex);
```

  and so the method defined could be:

```
void SomeMethodName(Exception ex);
```

  b) Instantiate this method as a delegate using something similar to the following example:

```
ExceptionListener OnExMethod = new ExceptionListener(SomeMethodName)
```

  c) Register the delegate with the connection by setting its ExceptionListener property:

```
connection.ExceptionListener = OnExMethod ;
```

  You can remove the delegate by resetting the ExceptionListener to:

```
null:  connection.ExceptionListener = null;
```

Automatic IBM MQ Client reconnection through XMS

Configure your XMS client to reconnect automatically following a network, queue manager, or server failure while using IBM WebSphere MQ 7.1 client and later, as the message provider.

Use the `WMQ_CONNECTION_NAME_LIST` and `WMQ_CLIENT_RECONNECT_OPTIONS` properties of the `MQConnectionFactory` class to configure a client connection to automatically reconnect. Automatic client reconnection reconnects a client after a connection failure, or as an option after stopping the queue manager. The design of some client applications makes them unsuitable for automatic reconnection.

Automatically reconnectable client connections become reconnectable once the connection is established.

Note: The properties `Client Reconnect Options`, `Client Reconnect Timeout`, and `Connection Namelist` can also be set via Client Channel Definitions Table (CCDT) or by enabling the client reconnection via the `mqclient.ini` file.

Note: If reconnection properties are set on the `ConnectionFactory` object and as well as in the CCDT, the precedence rule is as follows. If the default value of the connection name list property is set in the `ConnectionFactory` object, then the CCDT takes precedence. If the connection name list is not set to its default value, the property values set in the `ConnectionFactory` object take precedence. The default value of the connection namelist is `localhost(1414)`.

Working with XMS .NET administered objects

The topics in this section provide information about administered objects. XMS applications can retrieve object definitions from a central administered objects repository, and use them to create connection factories and destinations.

About this task

This section provides information to help with creating and managing administered objects, describing the types of administered object repository that XMS supports. The section also explains how an XMS application makes a connection to an administered objects repository to retrieve the required administered objects.

This section contains the following topics:

XMS .NET supported types of administered object repository

File System and LDAP administered objects can be used to connect to IBM MQ and WebSphere Application Server, whereas COS Naming can be used to connect only to the WebSphere Application Server.

File System object directories take the form of serialized Java Naming Directory Interface (JNDI) objects. LDAP object directories are directories that contain JNDI objects. File System and LDAP object directories can be administered by the IBM MQ Explorer, which is provided with IBM MQ and later. Both the File system and the LDAP object directories can be used to administer client connections by centralizing IBM MQ connection factories and destinations. The network administrator can deploy multiple applications that refer to the same central repository, and that are automatically updated to reflect changes to connection settings made in the central repository.

A COS naming directory contains WebSphere Application Server service integration bus connection factories and destinations and can be administered by using the WebSphere Application Server administrative console. For an XMS application to retrieve objects from the COS naming directory, a JNDI lookup web service must be deployed. This web service is not available on all WebSphere Application Server service integration technologies. Refer to the product documentation for details.

Note: Restart application connections for changes to the object directory to take effect.

XMS .NET property mapping for administered objects

To enable XMS .NET applications to use IBM MQ JMS and WebSphere Application Server connection factory and destination object definitions, the properties that are retrieved from these definitions must be mapped on to the corresponding XMS properties that can be set on XMS connection factories and destinations.

To create, for example, an XMS connection factory with properties that are retrieved from an IBM MQ JMS connection factory, the properties must be mapped between the two.

All property mappings are performed automatically.

Table 87 on page 636 demonstrates the mappings between some of the most common properties of connection factories and destinations. The properties that are shown in this table are just a small set of examples, and not all properties that are shown are relevant to all connection types and servers.

Table 87. Examples of name mapping for connection factory and destination properties

IBM MQ JMS property name	XMS property name	WebSphere Application Server service integration bus property name
PERSISTENCE (PER)	XMSC_DELIVERY_MODE	
EXPIRY (EXP)	XMSC_TIME_TO_LIVE	
PRIORITY (PRI)	XMSC_PRIORITY	
	XMSC_WPM_HOST_NAME	serverName
	XMSC_WPM_BUS_NAME	busName
	XMSC_WPM_TOPIC_SPACE	topicName

Note: The properties shown in Table 88 on page 636 are applicable for JMS as well as XMS .NET.

Table 88. XMS .NET properties

Property	Object type				
	CF	QCF	TCF	Queue	Topic
APPLICATIONNAME	Y	Y	Y	N/A	N/A
ASYNCEXCEPTION	Y	Y	Y	N/A	N/A
CCDTURL	Y	Y	Y	N/A	N/A
CHANNEL	Y	Y	Y	N/A	N/A
CONNECTIONNAMELIST	Y	Y	Y	N/A	N/A
CLIENTRECONNECTOPTIONS	Y	Y	Y	N/A	N/A
CLIENTRECONNECTTIMEOUT	Y	Y	Y	N/A	N/A
CLIENTID	N/A	Y	N/A	N/A	N/A
COMPHDR "1" on page 637	Y	N/A	Y	N/A	N/A
COMPMSG "1" on page 637	Y	Y	Y	N/A	N/A
CONNOPT "1" on page 637	Y	Y	Y	N/A	N/A
CONNTAG "1" on page 637	Y	Y	Y	N/A	N/A
DESCRIPTION "1" on page 637	N/A	Y	N/A	Y	Y

Table 88. XMS .NET properties (continued)					
Property	**Object type**				
	CF	**QCF**	**TCF**	**Queue**	**Topic**
EXPIRY "1" on page 637	N/A	N/A	N/A	Y	Y
FAILIFQUIESCE	Y	Y	Y	Y	Y
HOSTNAME	N/A	Y	N/A	N/A	N/A
LOCALADDRESS	N/A	Y	N/A	N/A	N/A
PERSISTENCE	N/A	N/A	N/A	Y	Y
PORT	N/A	Y	N/A	N/A	N/A
PRIORITY "1" on page 637	N/A	N/A	N/A	Y	Y
PROVIDERVERSION "1" on page 637	N/A	Y	N/A	N/A	N/A
QMANAGER	Y	Y	Y	Y	N/A
QUEUE "1" on page 637	N/A	N/A	N/A	Y	N/A
SHARECONVALLOWED	Y	Y	Y	N/A	N/A
TOPIC "1" on page 637	N/A	N/A	N/A	N/A	Y
TRANSPORT "1" on page 637	N/A	Y	N/A	N/A	N/A

Note:

1. These properties do not have application level properties but they can optionally be set using administered properties.

OutboundSNI property

From IBM MQ 9.3.0, you can set the XMSC_WMQ_OUTBOUND_SNI property, which sets the **OutboundSNI** property in an application.

The XMSC_WMQ_OUTBOUND_SNI_PROPERTY takes the following values:

- XMSC_WMQ_OUTBOUND_SNI_CHANNEL, which maps to "CHANNEL"
- XMSC_WMQ_OUTBOUND_SNI_HOSTNAME, which maps to "HOSTNAME"
- XMSC_WMQ_OUTBOUND_SNI_ASTERISK, which maps to "*"

Additionally, you can set the **OutboundSNI** property using the MQOUTBOUND_SNI environment variable, which takes the following values:

- CHANNEL
- HOSTNAME
- *

Note: The property defaults to XMSC_WMQ_OUTBOUND_SNI_CHANNEL if no specific value is set.

The order of precedence for setting the **OutboundSNI** property in the managed node is:

1. Application level property
2. Environment variable

For the **OutboundSNI** property in unmanaged node, `mqclient.ini` only is supported.

XMS .NET required properties for administered ConnectionFactory objects

When an application creates a connection factory, a number of properties must be defined to create a connection to a messaging server.

The properties listed in the following tables are the minimum required for an application to set to create a connection to a messaging server. If you want to customize the way that a connection is created, then your application can set any additional properties of the ConnectionFactory object as necessary. For more information, see Properties of ConnectionFactory. A complete list of available properties is included.

Connection to an IBM MQ queue manager

Table 89. Property settings for administered ConnectionFactory objects for connections to an IBM MQ queue manager

Required XMS property	Equivalent IBM MQ JMS property required
XMSC_CONNECTION_TYPE	XMS works this out from the connection factory class name and TRANSPORT (TRAN) property.
XMSC_WMQ_HOST_NAME	HOSTNAME (HOST)
XMSC_WMQ_PORT	PORT
XMSC_WMQ_QUEUE_MANAGER	Name of the Queue Manager

Real-time connection to a broker

Table 90. Property settings for administered ConnectionFactory objects for real-time connections to a broker

Required XMS	Equivalent IBM MQ JMS property required
XMSC_CONNECTION_TYPE	XMS works this out from the connection factory class name and TRANSPORT (TRAN) property.
XMSC_RTT_HOST_NAME	HOSTNAME (HOST)
XMSC_RTT_PORT	PORT

Connection to a WebSphere Application Server service integration bus

Table 91. Property settings for administered ConnectionFactory objects for connections to a WebSphere Application Server service integration bus

XMS property	Description
XMSC_CONNECTION_TYPE	The type of messaging server to which an application connects.. This is determined from the connection factory class name.
XMSC_WPM_BUS_NAME	For a connection factory, the name of the service integration bus that the application connects to or, for a destination, the name of the service integration bus in which the destination exists.

XMS .NET required properties for administered Destination objects

An application that is creating a destination must set several properties that the application on an administered Destination object.

Table 92. Property settings for administered Destination objects

Type of connection	Property	Description
IBM MQ queue manager	QUEUE (QU) TOPIC (TOP)	The queue that you want to connect to The topic that the application uses as a destination
Real-time connection to a broker	TOPIC (TOP)	The topic that the application uses as a destination
WebSphere Application Server service integration bus	topicName queueName	If your application is connecting to a topic If your application is connecting to a queue

XMS .NET creating administered objects

The ConnectionFactory and Destination object definitions that XMS applications require to make a connection to a messaging server must be created using the appropriate administrative tools.

Before you begin

For further details about the different types of administered object repository that XMS supports, see “XMS .NET supported types of administered object repository” on page 635.

About this task

To create the administered objects for IBM MQ use the IBM MQ Explorer or IBM MQ JMS administration (JMSAdmin) tool.

To create the administered objects for IBM MQ or IBM Integration Bus, use the IBM MQ JMS administration (JMSAdmin) tool.

To create administered objects for WebSphere Application Server service integration bus, use the WebSphere Application Server administrative console.

In the administrative tooling the property is known as **APPLICATIONNAME** or **APPNAME** for short.

Note: You cannot use JMSAdmin to set TRANSPORT(UNMANAGED). Therefore, in order to get an unmanaged XMS client using an administratively chosen application name, you need to enter the following command:

```
cf.SetIntProperty(XMSC.WMQ_CONNECTION_MODE, XMSC.WMQ_CM_CLIENT_UNMANAGED);
```

The following steps summarize what you do to create administered objects.

Procedure

1. Create a connection factory and define the necessary properties to create a connection from your application to your chosen server.

   The minimum properties that XMS requires to make a connection are defined in “XMS .NET required properties for administered ConnectionFactory objects” on page 638.
2. Create the required destination on the messaging server, which your application connects to:
   - For a connection to an IBM MQ queue manager, create a queue or topic.
   - For a real-time connection to a broker, create a topic.
   - For a connection to a WebSphere Application Server service integration bus, create a queue or a topic.

The minimum properties that XMS requires to make a connection are defined in "XMS .NET required properties for administered Destination objects" on page 639.

XMS .NET creating InitialContext objects

An application must create an initial context to be used to make a connection to the administered objects repository to retrieve the required administered objects.

About this task

An InitialContext object encapsulates a connection to the repository. The XMS API provides methods to perform the following tasks:

- Create an InitialContext object
- Look up an administered object in the administered object repository.

Procedure

- For further details about creating an InitialContext object, see InitialContext for .NET and Properties of InitialContext.

XMS .NET InitialContext properties

The parameters of the InitialContext constructor include the location of the repository of administered objects, given as a uniform resource indicator (URI). In order for an application to establish a connection to the repository, it may be necessary to provide more information than the information contained in the URI.

In JNDI and in the .NET implementation of XMS, the additional information is provided in an environment Hashtable to the constructor.

The location of the administered object repository is defined in the XMSC_IC_URL property. This property is typically passed on the Create call, but can be modified to connect to a different naming directory before the lookup. For FileSystem or LDAP contexts, this property defines the address of the directory. For COS naming, this is the address of the web service that uses these properties to connect to the JNDI directory.

The following properties are passed unmodified to the web service which will use them to use to connect to the JNDI directory.

- XMSC_IC_PROVIDER_URL
- XMSC_IC_SECURITY_CREDENTIALS
- XMSC_IC_SECURITY_AUTHENTICATION
- XMSC_IC_SECURITY_PRINCIPAL
- XMSC_IC_SECURITY_PROTOCOL

URI format for XMS initial contexts

The location of the repository of administered objects is provided as a uniform resource indicator (URI). The format of the URI depends on the context type.

FileSystem context

For the FileSystem context, the URL gives the location of the file system based directory. The structure of the URL is as defined by RFC 1738, *Uniform Resource Locators (URL)*: the URL has the prefix `file://`, and the syntax following this prefix is a valid definition of a file that can be opened on the system on which XMS is running.

This syntax can be platform-specific, and can use either '/ separators or '\' separators. If you use '\', then each separator needs to be escaped by using an additional '\'. This prevents the .NET framework from trying to interpret the separator as an escape character for what follows.

These examples illustrate this syntax:

```
file://myBindings
file:///admin/.bindings
file://\\admin\\.bindings
file://c:/admin/.bindings
file://c:\\admin\\.bindings
file://\\\\madison\\shared\\admin\\.bindings
file:///usr/admin/.bindings
```

LDAP context

For the LDAP context, the basic structure of the URL is as defined by RFC 2255, *The LDAP URL Format*, with the case-insensitive prefix `ldap://`

The precise syntax is illustrated in the following example:

```
LDAP://[Hostname][:Port]["/"[DistinguishedName]]
```

This syntax is as defined in the RFC but without support for any attributes, scope, filters, or extensions.

Examples of this syntax include:

```
ldap://madison:389/cn=JMSData,dc=IBM,dc=UK
ldap://madison/cn=JMSData,dc=IBM,dc=UK
LDAP:///cn=JMSData,dc=IBM,dc=UK
```

WSS context

For the WSS context, the URL is in the form of a web services endpoint, with the prefix `http://`.

Alternatively, you can use the prefix `cosnaming://` or `wsvc://`,

These two prefixes are interpreted as meaning that you are using a WSS context with the URL accessed over http, which enables the initial context type to be derived easily directly from the URL.

Examples of this syntax include the following:

```
http://madison.ibm.com:9080/xmsjndi/services/JndiLookup
cosnaming://madison/jndilookup
```

JNDI Lookup web service for XMS .NET

To access a COS naming directory from XMS, a JNDI Lookup web service must be deployed on a WebSphere Application Server service integration bus server. This web service translates the Java information from the COS naming service into a form that XMS applications can read.

The web service is provided in the enterprise archive file SIBXJndiLookupEAR.ear, located within the installation directory. For the current release of IBM MQ Message Service Client (XMS) for .NET, SIBXJndiLookupEAR.ear can be found in the *install_dir*`\java\lib` directory. This can be installed within a WebSphere Application Server service integration bus server by using either the administrative console or the wsaadmin scripting tool. Refer to the product documentation for further information on deploying web service applications.

To define the web service within XMS applications, you simply need to set the XMSC_IC_URL property of the InitialContext object to the web service endpoint URL. For example, if the web service is deployed on a server host called MyServer, an example of a web service endpoint URL:

```
wsvc://MyHost:9080/SIBXJndiLookup/services/JndiLookup
```

Setting the XMSC_IC_URL property allows InitialContext Lookup calls to invoke the web service at the defined endpoint, which in turn looks up the required administered object from the COS naming service.

.NET applications can use the web service. The server-side deployment is the same for XMS C, /C++ and, XMS .NET. XMS .NET invokes the web service directly through the Microsoft .NET Framework.

XMS .NET retrieval of administered objects

XMS retrieves an administered object from the repository using the address provided when the InitialContext object is created, or in the InitialContext properties.

Objects to be retrieved can have the following types of names:

- A simple name describing the Destination object, for example, a queue destination called SalesOrders
- A composite name, which can be made up of SubContexts, separated by '/', and it must end with the object name. An example of a composite name is "Warehouse/PickLists/DispatchQueue2" where Warehouse and Picklists are SubContexts in the naming directory, and DispatchQueue2 is the name of a Destination object.

Preventing applications from using a newer XMS version

By default, when a newer XMS version is installed, the applications using the previous version automatically switch to the newer version without having to recompile.However, you can prevent applications from using the newer version by setting an attribute in the application configuration file.

About this task

The multiple versions coexistence feature ensures that installation of a newer XMS version does not overwrite the previous XMS version. Instead, multiple instances of similar XMS .NET assemblies coexist in the Global Assembly Cache (GAC), but have different version numbers. Internally, the GAC uses a policy file to route the application calls to the latest version of XMS. Applications run without a need for recompilation and can use new features available in the newer XMS .NET version.

Procedure

- If an application is required to use the older XMS .NET version, set the `publisherpolicy` attribute to no in the application configuration file.

  Note: An application configuration file is a file with a name that consists of the name of the executable program to which the file relates, with the suffix .config. For example, the application configuration file for text.exe would have the name text.exe.config.

  At any time, however, all the applications of a system use the same version of XMS .NET.

Securing communications for XMS applications

This section provides information about setting up secure communications to enable XMS applications to connect via Secure Sockets Layer (SSL) to a WebSphere Application Server service integration bus messaging engine or IBM MQ queue manager.

About this task

The section contains the following topics:

- “Secure connections to an IBM MQ queue manager” on page 643

- "CipherSuite and CipherSpec name mappings for XMS connections to an IBM MQ queue manager" on page 643
- "Secure connections to a WebSphere Application Server service integration bus messaging engine" on page 644
- "CipherSuite and CipherSpec name mappings for connections to a WebSphere Application Server service integration bus" on page 645

Secure connections to an IBM MQ queue manager

To enable an XMS .NET application to make secure connections to an IBM MQ queue manager, the relevant properties must be defined in the ConnectionFactory object.

The protocol used in the encryption negotiation can be either Secure Sockets Layer (SSL) or Transport Layer Security (TLS), depending on which CipherSuite you specify in the ConnectionFactory object.

ConnectionFactory properties for connections using SSL to an IBM MQ queue manager, with a brief description, are shown in the following table:

Table 93. Properties of ConnectionFactory for connections to an IBM MQ queue manager via SSL

Name of property	Description
XMSC_WMQ_SSL_CERT_STORES	The locations of the servers that hold the certificate revocation lists (CRLs) to be used on an SSL connection to a queue manager.
XMSC_WMQ_SSL_CIPHER_SPEC	The name of the CipherSpec to be used on a secure connection to a queue manager.
XMSC_WMQ_SSL_CIPHER_SUITE	The name of the CipherSuite to be used on a TLS connection to a queue manager. The protocol used in negotiating the secure connection depends on the specified CipherSuite.
XMSC_WMQ_SSL_CRYPTO_HW	Configuration details for the cryptographic hardware connected to the client system.
XMSC_WMQ_SSL_FIPS_REQUIRED	The value of this property determines whether an application can or cannot use non-FIPS compliant cipher suites. If this property is set to true, only FIPS algorithms are used for the client-server connection.
XMSC_WMQ_SSL_KEY_REPOSITORY	The location of the key database file in which keys and certificates are stored.
XMSC_WMQ_SSL_KEY_RESETCOUNT	The KeyResetCount represents the total number of unencrypted bytes sent and received within an SSL conversation before the secret key is renegotiated.
XMSC_WMQ_SSL_PEER_NAME	The peer name to be used on an SSL connection to a queue manager.

CipherSuite and CipherSpec name mappings for XMS connections to an IBM MQ queue manager

The InitialContext translates between the JMSAdmin Connection Factory property SSLCIPHERSUITE and the XMS near-equivalent XMSC_WMQ_SSL_CIPHER_SPEC. A similar translation is necessary if you specify a value for XMSC_WMQ_SSL_CIPHER_SUITE but omit value for XMSC_WMQ_SSL_CIPHER_SPEC.

Table 94 on page 644 lists the available CipherSpecs and their JSSE CipherSuite equivalents.

Table 94. Available CipherSpecs and their JSSE CipherSuite equivalents	
CipherSpec	**Equivalent JSSE CipherSuite**
TLS_RSA_WITH_3DES_EDE_CBC_SHA	SSL_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA	SSL_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA	SSL_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_DES_CBC_SHA	SSL_RSA_WITH_DES_CBC_SHA

Note: Deprecated TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.

Secure connections to a WebSphere Application Server service integration bus messaging engine

To enable an XMS .NET application to make secure connections to a WebSphere Application Server service integration bus messaging engine, the relevant properties must be defined in the ConnectionFactory object.

XMS provides SSL and HTTPS support for connections to a WebSphere Application Server service integration bus. SSL and HTTPS provide secure connections for authentication and confidentiality.

Like WebSphere security, XMS security is configured with respect to JSSE security standards and naming conventions, which include the use of CipherSuites to specify the algorithms that are used when negotiating a secure connection. The protocol used in the encryption negotiation can be either SSL or TLS, depending on which CipherSuite you specify in the ConnectionFactory object.

Table 95 on page 644 lists the properties that must be defined in the ConnectionFactory object.

Table 95. Properties of ConnectionFactory for secure connections to a WebSphere Application Server service integration bus messaging engine	
Name of property	**Description**
XMSC_WPM_SSL_CIPHER_SUITE	The name of the CipherSuite to be used on a TLS connection to a WebSphere Application Server service integration bus messaging engine. The protocol used in negotiating the secure connection depends on the specified CipherSuite.
XMSC_WPM_SSL_KEYRING_LABEL	The certificate to be used when authenticating with the server.

The following is an example of ConnectionFactory properties for secure connections to a WebSphere Application Server service integration bus messaging engine:

```
cf.setStringProperty(XMSC_WPM_PROVIDER_ENDPOINTS, host_name:port_number:chain_name);
cf.setStringProperty(XMSC_WPM_SSL_KEY_REPOSITORY, key_repository_pathname);
cf.setStringProperty(XMSC_WPM_TARGET_TRANSPORT_CHAIN, transport_chain);
cf.setStringProperty(XMSC_WPM_SSL_CIPHER_SUITE, cipher_suite);
cf.setStringProperty(XMSC_WPM_SSL_KEYRING_STASH_FILE, stash_file_pathname);
```

Where chain_name should be set to either BootstrapTunneledSecureMessaging or BootstrapSecureMessaging, and port_number is the number of the port on which the bootstrap server listens for incoming requests.

The following is an example of ConnectionFactory properties for secure connections to a WebSphere Application Server service integration bus messaging engine with sample values inserted:

```
  /* CF properties needed for an SSL connection */
    cf.setStringProperty(XMSC_WPM_PROVIDER_ENDPOINTS,"localhost:7286:BootstrapSecureMessaging");
    cf.setStringProperty(XMSC_WPM_TARGET_TRANSPORT_CHAIN,"InboundSecureMessaging");
    cf.setStringProperty(XMSC_WPM_SSL_KEY_REPOSITORY,"C:\\Program Files\\IBM\\gsk7\\bin\
\XMSkey.kdb");
    cf.setStringProperty(XMSC_WPM_SSL_KEYRING_STASH_FILE,"C:\\Program Files\\IBM\\gsk7\\bin\
\XMSkey.sth");
    cf.setStringProperty(XMSC_WPM_SSL_CIPHER_SUITE,"SSL_RSA_EXPORT_WITH_RC4_40_MD5");
```

CipherSuite and CipherSpec name mappings for connections to a WebSphere Application Server service integration bus

Because IBM Global Security Kit (GSKit) uses CipherSpecs rather than CipherSuites, the JSSE-style CipherSuite names specified in the XMSC_WPM_SSL_CIPHER_SUITE property must be mapped to the GSKit-style CipherSpec names.

Table 96 on page 645 lists the equivalent CipherSpec for each recognized CipherSuite.

Table 96. Available CipherSuites and their equivalent CipherSpecs

CipherSuite	CipherSpec equivalent
TLS_RSA_WITH_DES_CBC_SHA	TLS_RSA_WITH_DES_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA	TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA	TLS_RSA_WITH_AES_256_CBC_SHA

Note: Deprecated TLS_RSA_WITH_3DES_EDE_CBC_SHA is deprecated. However, it can still be used to transfer up to 32 GB of data before the connection is terminated with error AMQ9288. To avoid this error, you need to either avoid using triple DES, or enable secret key reset when using this CipherSpec.

XMS messages

This section describes the structure and content of XMS messages and explains how applications process XMS messages.

This section contains the following topics:

- “Parts of an XMS message” on page 645
- “Header fields in an XMS message” on page 646
- “Properties of an XMS message” on page 646
- “The body of an XMS message” on page 649
- “Message selectors” on page 652
- “Mapping XMS messages onto IBM MQ messages” on page 653

Parts of an XMS message

An XMS message consists of a header, a set of properties, and a body.

Header

The header of a message contains fields, and all messages contain the same set of header fields. XMS and applications use the values of the header fields to identify and route messages. For more information about header fields, see “Header fields in an XMS message” on page 646.

Set of properties
The properties of a message specify additional information about the message. Although all messages have the same set of header fields, every message can have a different set of properties. For more information, see "Properties of an XMS message" on page 646.

Body
The body of a message contains application data. For more information, see "The body of an XMS message" on page 649.

An application can select which messages it wants to receive. By using message selectors, which specify the selection criteria. The criteria can be based on the values of certain header fields and the values of any of the properties of a message. For more information about message selectors, see "Message selectors" on page 652.

Header fields in an XMS message

To allow an XMS application to exchange messages with a WebSphere JMS application, the header of an XMS message contains the JMS message header fields.

The names of these header fields commence with the prefix JMS. For a description of the JMS message header fields, see the *Java Message Service Specification*.

XMS implements the JMS message header fields as attributes of a Message object. Each header field has its own methods for setting and getting its value. For a description of these methods, see IMessage. A header field is always readable and writable.

Table 97 on page 646 lists the JMS message header fields and indicates how the value of each field is set for a transmitted message. Some of the fields are set automatically by XMS when an application sends a message or, in the case of JMSRedelivered, when an application receives a message.

Table 97. JMS message header fields.]

Name of the JMS message header field	**How the value is set for a transmitted message (in the format *method* [*class*])**
JMSCorrelationID	Set JMSCorrelationID [Message]
JMSDeliveryMode	Send [MessageProducer]
JMSDestination	Send [MessageProducer]
JMSExpiration	Send [MessageProducer]
JMSMessageID	Send [MessageProducer]
JMSPriority	Send [MessageProducer]
JMSRedelivered	Receive [MessageConsumer]
JMSReplyTo	Set JMSReplyTo [Message]
JMSTimestamp	Send [MessageProducer]
JMSType	Set JMSType [Message]

Properties of an XMS message

XMS supports three kinds of message property: JMS defined properties, IBM defined properties, and application-defined properties.

An XMS application can exchange messages with a WebSphere JMS application because XMS supports the following predefined properties of a Message object:

- The same JMS-defined properties that WebSphere JMS supports. The names of these properties begin with the prefix JMSX.

- The same IBM-defined properties that WebSphere JMS supports. The names of these properties begin with the prefix JMS_IBM_.

Each predefined property has two names:

- A JMS name, for a JMS-defined property, or a WebSphere JMS name, for an IBM-defined property.

  This is the name by which the property is known in JMS or WebSphere JMS, and it is also the name that is transmitted with a message that has this property. An XMS application uses this name to identify the property in a message selector expression.
- An XMS name to identify the property in all situations except in a message selector expression. Each XMS name is defined as a named constant in `IBM.XMS.XMSC` class. The value of the named constant is the corresponding JMS or WebSphere JMS name.

In addition to the predefined properties, an XMS application can create and use its own set of message properties. These properties are called *application defined properties.*

After an application creates a message, the properties of the message are readable and writable. The properties remain readable and writable after the application sends the message. When an application receives a message, the properties of the message are read-only. If an application calls the `Clear Properties` method of the Message class when the properties of a message are read-only, the properties become readable and writable. The method also clears the properties.

The received message, when forwarded after clearing up the message properties, will behave in a manner consistent with the behavior of forwarding a standard WMQ XMS for .NET BytesMessage with message properties cleared up.

This is, however, not recommended since the following properties will be lost:

- JMS_IBM_Encoding property value, implying that the message data cannot be decoded meaningfully.
- JMS_IBM_Format property value, implying that the header chaining between the (MQMD or the new MQRFH2) message header and existing headers would be broken.

To determine the values of all the properties of a message, an application can call the Get Properties method of the Message class. The method creates an iterator that encapsulates a list of Property objects, where each Property object represents a property of the message. The application can then use the methods of the Iterator class to retrieve each Property object in turn, and it can use the methods of the Property class to retrieve the name, data type, and value of each property.

JMS-defined properties of a message

Several JMS-defined properties of a message are supported by both XMS and WebSphere JMS.

Table 98 on page 647 lists the JMS-defined properties of a message that are supported by both XMS and WebSphere JMS. For a description of the JMS-defined properties, see *Java Message Service Specification*. The JMS-defined properties are not valid for a real-time connection to a broker.

The table specifies the data type of each property and indicates how the value of the property is set for a transmitted message. Some of the properties are set automatically by XMS when an application sends a message or, in the case of JMSXDeliveryCount, when an application receives a message.

Table 98. JMS-defined properties of a message

XMS name of the JMS defined property	**JMS name**	**Data type**	**How the value is set for a transmitted message (in the format *method* [*class*])**
JMSX_APPID	JMSXAppID	System.String	Send [MessageProducer]
JMSX_DELIVERY_COUNT	JMSXDeliveryCount	System.Int32	Receive [MessageConsumer]
JMSX_GROUPID	JMSXGroupID	System.String	Set String Property [PropertyContext]

Table 98. JMS-defined properties of a message (continued)			
XMS name of the JMS defined property	**JMS name**	**Data type**	**How the value is set for a transmitted message (in the format *method* [*class*])**
JMSX_GROUPSEQ	JMSXGroupSeq	System.Int32	Set Integer Property [PropertyContext]
JMSX_USERID	JMSXUserID	System.String	Send [MessageProducer]

IBM-defined properties of a message

Several IBM-defined properties of a message are supported by XMS and WebSphere JMS.

Table 99 on page 648 lists the IBM defined properties of a message that are supported by both XMS and WebSphere JMS. For more information about the IBM-defined properties, see the IBM MQ or the WebSphere Application Server product documentation.

The table specifies the data type of each property and indicates how the value of the property is set for a transmitted message. Some of the properties are set automatically by XMS when an application sends a message.

Table 99. IBM-defined properties of a message			
XMS name of the IBM defined property	**WebSphere JMS name**	**Data type**	**How the value is set for a transmitted message (in the format *method* [*class*])**
JMS_IBM_CHARACTER_SET	JMS_IBM_Character_Set	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_ENCODING	JMS_IBM_Encoding	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_EXCEPTIONMESSAGE	JMS_IBM_ExceptionMessage	System.String	Receive [MessageConsumer]
JMS_IBM_EXCEPTIONREASON	JMS_IBM_ExceptionReason	System.Int32	Receive [MessageConsumer]
JMS_IBM_EXCEPTIONTIMESTAMP	JMS_IBM_ExceptionTimestamp	System.Int64	Receive [MessageConsumer]
JMS_IBM_EXCEPTIONPROBLEM DESTINATION	JMS_IBM_ExceptionProblemDestination	System.String	Receive [MessageConsumer]
JMS_IBM_FEEDBACK	JMS_IBM_Feedback	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_FORMAT	JMS_IBM_Format	System.String	Set String Property [PropertyContext]
JMS_IBM_LAST_MSG_IN_GROUP	JMS_IBM_Last_Msg_In_Group	System.Boolean	Set Integer Property [PropertyContext]
JMS_IBM_MSGTYPE	JMS_IBM_MsgType	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_PUTAPPLTYPE	JMS_IBM_PutApplType	System.Int32	Send [MessageProducer]
JMS_IBM_PUTDATE	JMS_IBM_PutDate	System.String	Send [MessageProducer]

Table 99. IBM-defined properties of a message (continued)

XMS name of the IBM defined property	WebSphere JMS name	Data type	How the value is set for a transmitted message (in the format *method* [*class*])
JMS_IBM_PUTTIME	JMS_IBM_PutTime	System.String	Send [MessageProducer]
JMS_IBM_REPORT_COA	JMS_IBM_Report_COA	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_COD	JMS_IBM_Report_COD	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_DISCARD_MSG	JMS_IBM_Report_Discard_Msg	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_EXCEPTION	JMS_IBM_Report_Exception	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_EXPIRATION	JMS_IBM_Report_Expiration	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_NAN	JMS_IBM_Report_NAN	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_PAN	JMS_IBM_Report_PAN	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_PASS_CORREL_ ID	JMS_IBM_Report_Pass_Correl_ID	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_REPORT_PASS_MSG_ID	JMS_IBM_Report_Pass_Msg_ID	System.Int32	Set Integer Property [PropertyContext]
JMS_IBM_SYSTEM_MESSAGEID	JMS_IBM_System_MessageID	System.String	Send [MessageProducer]

Application-defined properties of a message

An XMS application can create and use its own set of message properties. When an application sends a message, these properties are also transmitted with the message. A receiving application, using message selectors, can then select which messages it wants to receive based on the values of these properties.

To allow a WebSphere JMS application to select and process messages sent by an XMS application, the name of an application-defined property must conform to the rules for forming identifiers in message selector expressions. for more information, see “Message selectors in JMS” on page 139. The value of an application-defined property must have one of the following data types: System.Boolean, System.SByte, System.Int16, System.Int32, System.Int64, System.Float, System.Double, or System.String.

The body of an XMS message

The body of a message contains application data. However, a message can have no body, and comprise only the header fields and properties.

XMS supports five types of message body:

Bytes
The body contains a stream of bytes. A message with this type of body is called a *bytes message*. The IBytesMessage interface contains the methods to process the body of a bytes message.

Map
The body contains a set of name-value pairs, where each value has an associated data type. A message with this type of body is called a *map message*. The IMapMessage interface contains the methods to process the body of a map message.

Object
The body contains a serialized Java or .NET object. A message with this type of body is called an *object message*. The IObjectMessage interface contains the methods to process the body of an object message.

Stream
The body contains a stream of values, where each value has an associated data type. A message with this type of body is called a *stream message*. The IStreamMessage interface contains the methods to process the body of a stream message.

Text
The body contains a string. A message with this type of body is called a *text message*. The ITextMessage interface contains the methods to process the body of a text message.

The IMessage interface is the parent of all message objects and can be used in messaging functions to represent any of the XMS message types.

For information about the size and maximum and minimum values of each of these data types, see Table 85 on page 630.

Bytes messages

The body of a bytes message contains a stream of bytes. The body contains only the actual data, and it is the responsibility of the sending and receiving applications to interpret this data.

Bytes messages are useful if an XMS application needs to exchange messages with applications that are not using the XMS or JMS application programming interface.

After an application creates a bytes message, the body of the message is write-only. The application assembles the application data into the body by calling the appropriate write methods of the IBytesMessage interface for .NET. Each time the application writes a value to the bytes message stream, the value is assembled immediately after the previous value written by the application. XMS maintains an internal cursor to remember the position of the last byte that was assembled.

When the application sends the message, the body of the message becomes read-only. In this mode, the application can send the message repeatedly.

When an application receives a bytes message, the body of the message is read-only. The application can use the appropriate read methods of the IBytesMessage interface to read the contents of the bytes message stream. The application reads the bytes in sequence, and XMS maintains an internal cursor to remember the position of the last byte that was read.

If an application calls the Reset method of the IBytesMessage interface when the body of a bytes message is writeable, the body becomes read-only. The method also repositions the cursor at the beginning of the bytes message stream.

If an application calls the `Clear Body` method of the IMessage interface for .NET when the body of a bytes message is read-only, the body becomes writeable. The method also clears the body.

Map messages

The body of a map message contains a set of name-value pairs, where each value has an associated data type.

In each name-value pair, the name is a string that identifies the value, and the value is an element of application data that has one of the XMS data types listed in Table 100 on page 652. The order of the name-value pairs is not defined. The MapMessage class contains the methods to set and get name-value pairs.

An application can access a name-value pair randomly by specifying its name.

A .NET application can use the MapNames property to get an enumeration of the names in the body of the map message.

When an application gets the value of a name-value pair, the value can be converted by XMS into another data type. For example, to get an integer from the body of a map message, an application can call the GetString method of the MapMessage class, which returns the integer as a string. The supported conversions are the same as those that are supported when XMS converts a property value from one data type to another. For more information about the supported conversions, see “Implicit conversion of a property value from one data type to another” on page 630.

After an application creates a map message, the body of the message is readable and writable. The body remains readable and writable after the application sends the message. When an application receives a map message, the body of the message is read-only. If an application calls the Clear Body method of the Message class when the body of a map message is read-only, the body becomes readable and writable. The method also clears the body.

Object messages

The body of an object message contains a serializedJava or .NET object.

An XMS application can receive an object message, change its header fields and properties, and then send it to another destination. An application can also copy the body of an object message and use it to form another object message. XMS treats the body of an object message as an array of bytes.

After an application creates an object message, the body of the message is readable and writable. The body remains readable and writable after the application sends the message. When an application receives an object message, the body of the message is read-only. If an application calls the `Clear Body` method of the IMessage interface for .NET when the body of an object message is read-only, the body becomes readable and writable. The method also clears the body.

Stream messages

The body of a stream message contains a stream of values, where each value has an associated data type.

The data type of a value is one of the XMS data types listed in Table 100 on page 652.

After an application creates a stream message, the body of the message is writable. The application assembles the application data into the body by calling the appropriate write methods of the IStreamMessage interface for .NET. Each time the application writes a value to the message stream, the value, and its data type are assembled immediately after the previous value written by the application. XMS maintains an internal cursor to remember the position of the last value that was assembled.

When the application sends the message, the body of the message becomes read-only. In this mode, the application can send the message multiple times.

When an application receives a stream message, the body of the message is read-only. The application can use the appropriate read methods of the IStreamMessage interface for .NET to read the contents of the message stream. The application reads the values in sequence, and XMS maintains an internal cursor to remember the position of the last value that was read.

When an application reads a value from the message stream, the value can be converted by XMS into another data type. For example, to read an integer from the message stream, an application can call the ReadString method, which returns the integer as a string. The supported conversions are the same as those that are supported when XMS converts a property value from one data type to another. For more information about the supported conversions, see “Implicit conversion of a property value from one data type to another” on page 630.

If an error occurs while an application is attempting to read a value from the message stream, the cursor is not advanced. The application can recover from the error by attempting to read the value as another data type.

If an application calls the Reset method of the IStreamMessage interface for XMS when the body of a stream message is write-only, the body becomes read-only. The method also repositions the cursor at the beginning of the message stream.

If an application calls the `Clear Body` method of the IMessage interface for XMS when the body of a stream message is read-only, the body becomes write-only. The method also clears the body.

Text messages

The body of a text message contains a string.

After an application creates an text message, the body of the message is readable and writable. The body remains readable and writable after the application sends the message. When an application receives an text message, the body of the message is read-only. If an application calls the Clear Body method of the IMessage interface for .NET when the body of an text message is read-only, the body becomes readable and writable. The method also clears the body.

Data types for elements of application data

To ensure that an XMS application can exchange messages with an IBM MQ classes for JMS application, both the applications must be able to interpret the application data in the body of a message in the same way.

For this reason, each element of application data written in the body of a message by an XMS application must have one of the data types listed in Table 100 on page 652. For each data type, the table shows the compatible Java data type. XMS provides the methods to write elements of application data only with these data types.

Table 100. XMS data types that are compatible with Java data types

XMS Data type	Represents	Compatible Java data type
System.Boolean	The boolean value true or false	boolean
System.Char16	Double byte character	char
System.SByte	Signed 8-bit integer	byte
System.Int16	Signed 16-bit integer	short
System.Int32	Signed 32-bit integer	int
System.Int64	Signed 64-bit integer	long
System.Float	Signed floating point number	float
System.Double	Signed double precision floating point number	double
System.String	String of characters	String

For information about the size, maximum value and minimum value of each of these data types, see “XMS primitive types” on page 630.

Message selectors

An XMS application uses messages selectors to select the messages it wants to receive.

When an application creates a message consumer, it can associate a message selector expression with the consumer. The message selector expression specifies the selection criteria.

When an application is connecting to IBM WebSphere MQ 7.0 queue manager the message selection is done at the queue manager side. XMS does not do any selection and simply delivers the message it received from the queue manager thus providing better performance.

An application can create more than one message consumer, each with its own message selector expression. If an incoming message meets the selection criteria of more than one message consumer, XMS delivers the message to each of these consumers.

A message selector expression can reference the following properties of a message:

- JMS-defined properties
- IBM-defined properties
- Application-defined properties

It can also reference the following message header fields:

- JMSCorrelationID
- JMSDeliveryMode
- JMSMessageID
- JMSPriority
- JMSTimestamp
- JMSType

A message selector expression, however, cannot reference data in the body of a message.

Here is an example of a message selector expression:

```
JMSPriority > 3 AND manufacturer = 'Jaguar' AND model in ('xj6','xj12')
```

XMS delivers a message to a message consumer with this message selector expression only if the message has a priority greater than 3; an application-defined property, manufacturer, with a value of `Jaguar;` and another application defined-property, model, with a value of `xj6` or `xj12.`

The syntax rules for forming a message selector expression in XMS are the same as those in IBM MQ classes for JMS. For information about how to construct a message selector expression, see the IBM MQ product documentation Note that, in a message selector expression, the names of JMS-defined properties must be the JMS names, and the names of IBM-defined properties must be the IBM MQ classes for JMS names. You cannot use the XMS names in a message selector expression.

Mapping XMS messages onto IBM MQ messages

The JMS header fields and properties of an XMS message are mapped onto fields in the header structures of an IBM MQ message.

When an XMS application is connected to an IBM MQ queue manager, messages sent to the queue manager are mapped onto IBM MQ messages in the same way that IBM MQ classes for JMS messages are mapped onto IBM MQ messages in similar circumstances.

If the XMSC_WMQ_TARGET_CLIENT property of a Destination object is set to XMSC_WMQ_TARGET_DEST_JMS, the JMS header fields and properties of a message sent to the destination are mapped onto fields in the MQMD and MQRFH2 header structures of the IBM MQ message. Setting the XMSC_WMQ_TARGET_CLIENT property in this way assumes that the application that receives the message can handle an MQRFH2 header. The receiving application might therefore be another XMS application, an IBM MQ classes for JMS application, or a native IBM MQ application that has been designed to handle an MQRFH2 header.

If the XMSC_WMQ_TARGET_CLIENT property of a Destination object is set to XMSC_WMQ_TARGET_DEST_MQ instead, the JMS header fields and properties of a message sent to the destination are mapped onto fields in the MQMD header structure of the IBM MQ message. The message does not contain an MQRFH2 header, and any JMS header fields and properties that cannot be mapped onto fields in the MQMD header structure are ignored. The application that receives the message can therefore be a native IBM MQ that's not designed to handle an MQRFH2 header.

IBM MQ messages received from a queue manager are mapped onto XMS messages in the same way that IBM MQ messages are mapped onto IBM MQ classes for JMS messages in similar circumstances.

If an incoming IBM MQ message has an MQRFH2 header, the resulting XMS message has a body whose type is determined by the value of the **Msd** property contained in the `mcd` folder of the MQRFH2 header. If the **Msd** property is not present in the MQRFH2 header, or if the IBM MQ message has no MQRFH2 header, the resulting XMS message has a body whose type is determined by the value of the *Format* field in the MQMD header. If the *Format* field is set to MQFMT_STRING, the XMS message is a text message. Otherwise, the XMS message is a bytes message. If the IBM MQ message has no MQRFH2 header, only those JMS header fields and properties that can be derived from fields in the MQMD header are set.

For more information about mapping IBM MQ classes for JMS messages onto IBM MQ messages, see "Mapping JMS messages onto IBM MQ messages" on page 142.

Reading and writing the message descriptor from a IBM MQ Message Service Client (XMS) for .NET application

You can access all the message descriptor (MQMD) fields of an IBM MQ message except StrucId and Version; BackoutCount can be read but not written to.

The message attributes provided by the IBM MQ Message Service Client (XMS) for .NET facilitates XMS applications to set MQMD fields and also to drive IBM WebSphere MQ applications.

Some restrictions apply when using publish/subscribe messaging. For example, MQMD fields like MsgID and CorrelId, if set, are ignored.

The function is also unavailable when the **PROVIDERVERSION** property is set to 6.

Accessing IBM MQ Message data from a IBM MQ Message Service Client (XMS) for .NET application

You can access the complete IBM MQ message data including the MQRFH2 header (if present) and any other IBM MQ headers (if present) within a IBM MQ Message Service Client (XMS) for .NET application as the body of a JMSBytesMessage.

The function described in this topic is available only when connecting to an IBM WebSphere MQ 7.0 or later queue manager and the IBM MQ messaging provider is in normal mode.

Destination object properties determine how the XMS application accesses the whole of an IBM MQ message (including the MQRFH2 header, if present) as the body of a JMSBytesMessage.

ALW Developing AMQP client applications

The IBM MQ support for AMQP APIs, allows an IBM MQ administrator to create an AMQP channel. When it is started, this channel defines a port number that accepts connections from AMQP client applications.

You can install an AMQP channel on AIX, Linux, and Windows systems; it is not available on IBM i or z/OS.

An AMQP 1.0 client application can connect to queue manager with an AMQP channel.

Developing applications using the Apache Qpid JMS library

Introduction

The Apache Qpid JMS library uses the AMQP 1.0 protocol to provide an implementation of the JMS 2 specification.

Apache Qpid JMS uses some aspects of the AMQP 1.0 protocol in a different way from the MQ Light messaging APIs. IBM MQ 9.2 added support to IBM MQ AMQP channels, so that Apache Qpid JMS applications are able to connect to IBM MQ and do publish/subscribe messaging, including the use of shared subscriptions.

IBM MQ 9.3 added further support to IBM MQ AMQP channels, so that Apache Qpid JMS applications are able to connect to IBM MQ and perform point to point messaging. See "Point-to-point support on AMQP channels" on page 659 for more information.

IBM MQ 9.3.0 added further queue browse support for IBM MQ AMQP channels, so that Apache Qpid JMS applications can connect to IBM MQ and perform browsing of messages from a queue. See "Point-to-point support on AMQP channels" on page 659 for more information.

IBM MQ 9.3.0 added two additional channel attributes for AMQP channels, TMPMODEL and TMPQPRFX. These attributes are for the model queue, and temporary queue prefix to be used while creating a temporary queue.

Intercommunication with other IBM MQ applications

It is possible to send messages between Apache Qpid JMS applications and other IBM MQ applications. For example, an Apache Qpid application can publish messages on a topic, and MQ Light applications can receive them by creating a subscription.

An Apache Qpid JMS application can also publish messages that are consumed by traditional IBM MQ applications, for example using the MQSUB API call to subscribe to the same topic.

Similarly, Apache Qpid JMS applications can subscribe to IBM MQ topics that traditional IBM MQ applications publish messages on.

It is also possible for an Apache Qpid JMS application to share a subscription with an MQ Light application, as long as both clients specify the same share name and topic pattern.

Note that, in order to do this, the Apache Qpid JMS application must not connect with a client ID. This ensures that the IBM MQ subscription name used by both of the applications is the same.

Attention: It is not possible for an Apache Qpid JMS application to share a subscription with an IBM MQ JMS application.

Apache Qpid JMS restrictions

The following JMS capabilities are supported:

- Client acknowledge , auto-acknowledge, and dups ok acknowledge mode (DUPS_OK_ACKNOWLEDGE)
  - Connecting with or without credentials
  - Creating a consumer on a topic destination
  - Creating a durable consumer on a topic destination
  - Creating a shared consumer on a topic destination
  - Creating a shared durable consumer on a topic destination
  - Client acknowledge and auto-acknowledge modes
  - Message acknowledgment and session acknowledgment
  - Unsubscribing from a durable subscription
  - Creating a temporary queue
  - Creating a consumer on a queue or temporary queue destination
  - JMS MessageListeners
  - JMS Consumer to receive body; the JMS 2.0 method called `Consumer.receiveBody()`
  - The following JMS message types are supported:
    - BytesMessage
    - MapMessage
    - ObjectMessage
    - StreamMessage
    - TextMessage
  - Browsing messages from a queue

The following JMS capabilities are not supported by AMQP clients:

- The use of transacted sessions and transacted JMSContexts
  - The use of message selectors
  - The use of the **nolocal** attribute
  - The use of transacted sessions
  - The use of delivery delay
  - At IBM MQ 9.3.0, browsing messages from a queue.
  - Creating multiple durable subscriptions or consumers with the same client ID and topic
  - JMS Temporary Topics
  - AMQP filters are not supported.

Downloading sample AMQP clients

IBM MQ does not ship AMQP clients, but you can download MQ Light clients or download open-source AMQP clients based on Apache Qpid libraries. For more information, see IBM MQ Light and Apache Qpid.

You can also download other open-source AMQP clients based on Apache Qpid libraries. For more information, see https://qpid.apache.org/index.html.

Attention: IBM Support is unable to provide configuration or defect support for these client packages, and any usage questions or code defect reports should be directed to the respective projects.

Deploying AMQP clients to IBM MQ

When an application is ready to deploy, it requires all of the monitoring, reliability, and security capabilities of other enterprise applications. It can also exchange data with other enterprise applications.

When you have deployed an AMQP client, you can exchange messages with IBM MQ applications. For example, if you use the AMQP client to send a JavaScript string message, the IBM MQ application receives an MQ message, where the format field of the MQMD is set to MQSTR.

Managing the AMQP channel

The AMQP channel can be managed in the same way as other MQ channels. You can use MQSC commands, PCF command messages, or IBM MQ Explorer to define, start, stop, and manage the channels. In Creating and using AMQP channels, example commands are provided to define and start connecting clients to a queue manager.

When an AMQP channel is started, you can test it by connecting an AMQP 1.0 client. For example, MQ Light, Apache Qpid Proton, or Apache Qpid JMS.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW MQ Light, Apache Qpid JMS, and AMQP (Advanced Message Queuing Protocol)

The MQ Light client, Apache Qpid clients like Apache Proton, and Apache Qpid JMS APIs are based on the OASIS Standard AMQP 1.0 wire protocol. AMQP specifies how messages are sent between senders and receivers. An application acts as a sender when the application sends a message to message broker, such as IBM MQ. IBM MQ acts as a sender when it sends a message to an AMQP application.

Some of the benefits of AMQP are as follows:

- An open standardized protocol
- Compatibility with other open source AMQP 1.0 clients

- Many open source client implementations available

Although any AMQP 1.0 client can connect to an AMQP channel, some AMQP features are not supported, for example transactions or multiple sessions.

For more information, see AMQP.org website and OASIS Standard AMQP 1.0 PDF.

The MQ Light and Apache Qpid JMS APIs have the following messaging features:

- At-most-once message delivery
- At-least-once message delivery
- Topic string destination addressing
- Message and destination durability
- Shared destinations to allow multiple subscribers to share workload
- Client takeover for easy resolution of hung clients
- Configurable read ahead of messages
- Configurable acknowledgment of messages

For complete documentation of the Apache Qpid JMS API, see Qpid JMS.

Related tasks
Creating and using AMQP channels
Securing AMQP clients

ALW AMQP 1.0 support

AMQP channels provide a level of support for AMQP 1.0-compliant applications.

AMQP channels support a subset of the AMQP 1.0 protocol. You can connect AMQP 1.0 compatible clients to an IBM MQ AMQP channel. To use all of the messaging features supported by AMQP channels, you must correctly set the value of certain AMQP 1.0 fields.

This information outlines the way AMQP fields must be formatted and lists the features of the AMQP 1.0 specification that are not supported by AMQP channels.

The following features of the AMQP 1.0 specification are either not supported or are limited in their use:

ATTACH frame

AMQP channels expect the capabilities in ATTACH frame to contain one of the following:

```
topic
temporary queue
queue
shared
```

Capabilities imply the type of object, and in the case of multi-capabilities, the order of priority of selecting the capability is topic, temporary-queue, queue.

If a capability does not contain an expected value, the default capability is topic. Any other capabilities are ignored.

Note: Some AMQP clients do not set these capabilities and will get the IBM MQ default behavior of publish/subscribe. For example, the Quarkus Reactive Messaging AMQP 1.0 Connector only sets capabilities from version 2.8.0CR1 onwards.

AMQP channels expect the `distribution-Mode` on the ATTACH frame to contain one of the following, for a source or target:

- `move`
- `copy`

where `move` implies a destructive get, and `copy` implies browser.

Note: If the `distribution-Mode` is not set, or set to anything other than `copy`, `move` is assumed.

Link names

AMQP channels expect the name of an AMQP link to follow one of these formats:

- A plain topic (for publishing and subscribing)
  - Publishing messages: a plain topic string (for example, a link name of "`/sports/football`") causes a message to be published on the `/sports/football` topic.
  - Subscribing to a topic to receive messages: a plain topic string (for example, a link name of "`/sports/football`" causes a subscription to be defined on the `/sports/football` topic.
- A private verbose topic (for subscribing)
  - A verbose topic string that describes a private subscription in the form: "`private:`*`topic string`*" (for example: "`private:/sports/football`"). The behavior is identical to a plain topic string. The `private` declaration differentiates a subscription specific to a particular AMQP client from a subscription shared between clients.
- A shared verbose topic (for subscribing)
  - A verbose topic string that describes a shared subscription in the form: "`share:`*`share name`*`:`*`topic string`*" (for example: "`share:bbc:/sports/football`").
- A queue (for point to point messaging for producer and consumer)
  - Producer to send messages; a queue name string causes a producer to send a message on a queue.
  - Consumer to receive messages; a queue name string causes a consumer to receive messages from a queue.
- Blank (for point to point messaging on a temporary queue)
  - Producer to send messages on a temporary queue; Blank causes a producer to send a message on a temporary queue.
  - Consumer to receive messages on a temporary queue; Blank causes a consumer to receive messages from a temporary queue.

For more information about the way AMQP messages are mapped to and from IBM MQ messages see “Mapping AMQP fields onto IBM MQ fields (incoming messages)” on page 663.

Maximum lengths for topic strings, share names, and client IDs

The topic string, share name, and client ID must be contained within 10237 bytes. In addition, the maximum length of a client ID is 256 characters.

These maximum lengths mean that you can have one of the following:

- a very long topic string, provided that the share name is short
- a long share name, but a short topic string

Container IDs

AMQP channels expect the container-id of an AMQP Open performative to contain a unique AMQP client ID. The maximum length of an AMQP client ID is 256 characters and the ID can contain alphanumeric characters, percent sign (%), slash (/), period (.) and underscore (_).

Sessions

AMQP channels support only a single AMQP session. An AMQP client that attempts to create more than one AMQP session receives an error message and is disconnected from the channel.

Transactions

AMQP channels do not support AMQP transactions. An AMQP attach frame that attempts to coordinate a new transaction or an AMQP transfer frame that attempts to declare a new transaction is rejected with an error message.

Delivery state

AMQP channels only support a delivery state for disposition frames of Accepted, Released, or Modified. Note that, where a state of Modified is used, AMQP channels do not support the undeliverable-here option.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW Point-to-point support on AMQP channels

The IBM MQ AMQP channel provides support for sending messages to queues and receiving messages from queues.

AMQP clients like an Apache Qpid™ JMS library request a `queue` or `temporary-queue` capability when sending the AMQP attach frame. Capabilities allow the AMQP channel to identify the object as a queue, temporary-queue or topic. In the absence of either a queue or temporary-queue capability, or even any of the capabilities, the request is assumed to be for a topic.

IBM MQ AMQP channels provide queue type support for the following:

Queue receive and send
Messages can be sent to a queue and consumed from a queue. For consuming messages, both synchronous and asynchronous modes are supported.

Queue browse message
As well as putting messages to a queue and getting messages from a queue, messages can also be browsed from a queue.

Temporary queue support
Messages can be sent to a temporary queue and consumed from a temporary queue. Note that temporary queue deletion is supported, if the same temporary queue object used to create the temporary queue is also used to delete the temporary queue.

The SYSTEM.DEFAULT.MODEL.QUEUE is used when creating a temporary queue, and the prefix for the temporary queue will be `AMQP.*`.

The SYSTEM.DEFAULT.MODEL.QUEUE is by default a temporary dynamic queue, but you can use the **`Definition type`** property on the SYSTEM.DEFAULT.MODEL.QUEUE queue to change the queue to be a permanent dynamic queue.

Permanent dynamic queue
A permanent dynamic queue is deleted when an AMQP client, such as an Apache Qpid JMS library, sends a request with a `detach` frame with the **`closed`** attribute set to *true*.

Important:

Qpid JMS behavior:
You must call a Qpid JMS API command, for example, the `javax.jms.TemporaryQueue.delete()` method to destroy the queue after use and this process also clears the messages present on the queue.

If you do not issue such a command, the queue remains with any messages still present, when the connection is closed.

-

Temporary dynamic queue
A temporary dynamic queue is deleted when AMQP client closes the connection.

Important:

Qpid JMS behavior:
If you call a Qpid JMS API command, for example, the `javax.jms.TemporaryQueue.delete()` method, close the JMS connection, or the connection breaks, the queue is deleted and any messages are lost.

Closing a JMS session in itself does not cause the temporary queue to be deleted, even though the temporary queue could have been created using the `javax.jms.Session.createTemporaryQueue()` method.

-

Related tasks
Creating and using AMQP channels
Securing AMQP clients

ALW Mapping AMQP and IBM MQ message fields

AMQP messages are composed of a header, delivery annotations, message annotations, properties, application properties, body, and footer.

AMQP messages are composed of the following parts:

Header
The optional header contains five fixed attributes of the message:

- **durable** - specifies durability requirements
- **priority** - relative message priority
- **ttl** - time to live in milliseconds
- **first-acquirer** - if this is true, the message has not been acquired by any other link
- **delivery-count** - the number of previous, unsuccessful delivery attempts.

Delivery-annotations
Optional. Specifies non-standard header attributes of the message for different intended audiences. Delivery annotations convey information from the sending peer to the receiving peer.

Message-annotations
Optional. Specifies non-standard header attributes of the message for different intended audiences. The message-annotations section is used for properties of the message which are aimed at the infrastructure and should be propagated across every delivery step.

Properties
Optional. This part is equivalent the MQ message descriptor. It contains the following fixed fields:

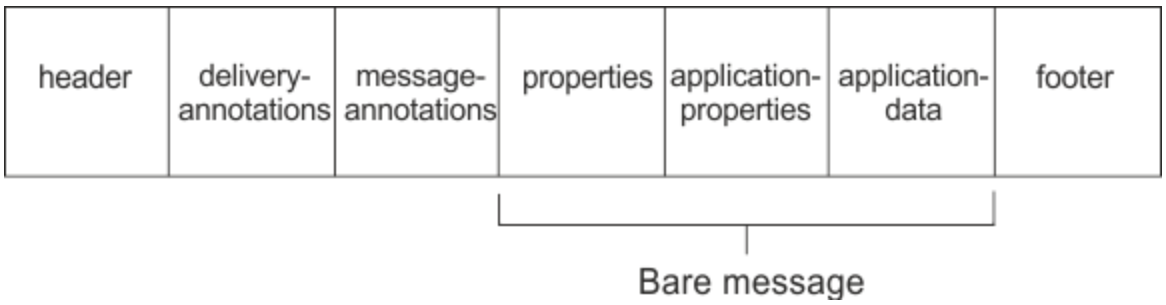
- **message-id** - application message identifier
- **user-id** - ID of creating user
- **to** - address of node that the message is destined for
- **subject** - the subject of the message
- **reply-to** - the node that the send replies to
- **correlation-id** - application correlation identifier
- **content-type** - MIME content type
- **content-encoding** - MIME content type. Used as a modifier to the content-type.
- **absolute-expiry-time** - the time when this message is considered expired
- **creation-time** - the time when this message was created
- **group-id** - the group that this message belongs to
- **group-sequence** - the sequence number of this message within its group
- **reply-to-group-id** - the group that the reply message belongs to

Applications-properties
Equivalent to MQ message properties.

Body
Equivalent to the MQ user payload.

Footer
Optional. The footer is used for details about the message or delivery that can only be calculated or evaluated after the whole bare message has been constructed or seen (for example, message hashes, HMACs, signatures and encryption details).

The AMQP message format is illustrated in the following figure:



The properties, application-properties, and application-data part are known as the "bare message". This is the message as sent by the sender, and is immutable. The receiver sees the entire message, including the header, footer, delivery-annotations and message-annotations.

For a full description of the AMQP 1.0 message format, see the OASIS Standard at https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf.

Related tasks
Creating and using AMQP channels
Securing AMQP clients

ALW Mapping IBM MQ fields onto AMQP fields (outgoing messages)

When an IBM MQ message is published and IBM MQ sends it to an AMQP consumer, it will propagate some of the attributes of the IBM MQ message into equivalent AMQP message attributes.

header

A header is only included if one of the five fields in the header contains a non-default value. Only the fields with a non-default value are included in the header. The five header fields are initially derived from the equivalent `mq_amqp.Hdr` property, if it is set, and then modified as shown in the following table:

Table 101. Header field mappings

Field	Default value	Value
durable	false	True if `MQMD.Persistence` is set to `MQPER_PERSISTENT`, false otherwise.
priority	4	From `mq_amqp.Hdr.Pri`, if set, or otherwise from `MQMD.Priority`, if set. If neither set, set to 4.
ttl	n/a	`MQMD.Expiry` in milliseconds. If the value of `MQMD.Expiry` is MQEI_UNLIMITED, then set to the maximum value for the AMQP ttl field
first-acquirer	false	From `mq_amqp.Hdr.Fac`, if set, or false otherwise.

Table 101. Header field mappings (continued)		
Field	**Default value**	**Value**
delivery-count	0	From `mq_amqp.Hdr.Dct`, if set, or 0 otherwise.

delivery-annotation

Set as necessary by the AMQP channel.

message-annotation

Not included.

properties

The **properties** will come unmodified from the equivalent `mq_amqp.Prp` properties if these are set. If the message was not originally an AMQP message (that is, PutApplType is not MQAT_AMQP), then a properties section is generated as described in the following table:

Table 102. Properties field mappings	
Name	**Value**
message-id	The `MQMD.MsgId` is set as binary.
user-id	The UTF-8 form of the `MQMD.UserIdentifier` is set as binary in network byte-order.
to	The queue that the message was got from, or, for a publication, the topic string.
subject	Not set.
reply-to	The `MQMD.ReplyToQ` if non-blank, otherwise not set.
correlation-id	The `MQMD.CorrelId` is set as binary if non-blank, otherwise not set.
content-type	Not set.
content-encoding	Not set.
absolute-expiry-time	Not set.
creation-time	The `MQMD.PutDate` and `MQMD.PutTime` fields are used to generate a timestamp.
group-id	Not set.
group-sequence	Not set.
reply-to-group-id	Not set.

application-properties

All IBM MQ properties in the "usr" group are added as the **application-properties**.

body

The AMQP channel performs a get with convert, to convert the IBM MQ payload into UTF-8.

If the IBM MQ payload does not contain an AMQP message, then the IBM MQ payload is set in the body as a single string data section for Format MQFMT_STRING (provided conversion to UTF-8 was successful), or as a single binary data section otherwise.

If an AMQP format message is included, then this is set as the body. Any IBM MQ headers (not including the messages properties which are returned in a message handle) that precede the AMQP message are prepended as a binary value if the body is an AMQP Sequence. Otherwise the IBM MQ headers are discarded.

footer

No footer is included.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

Related reference

MQMD - Message descriptor

ALW Mapping AMQP fields onto IBM MQ fields (incoming messages)

When the AMQP channel receives a message and puts it to IBM MQ, it propagates some of the attributes of the AMQP message into equivalent IBM MQ message attributes.

The following restrictions apply when mapping an incoming AMQP message:

- If the `message-id` or `correlation-id` field in the properties part is a uuid or a ulong, then the message is rejected.
- Any `message-annotations` cause the message to be rejected.
- `delivery-annotations` and `footer` sections are allowed, but are not propagated into the IBM MQ message.

The following sub-sections show the IBM MQ expression of an AMQP message.

Message descriptor

Table 103. Message descriptor for AMQP message

Field	Value
StrucId	MQMD_STRUC_ID
Version	MQMD_VERSION_1
Report	MQRO_NONE
MsgType	MQMT_DATAGRAM
Expiry	Value taken from the `ttl` field in the AMQP message header
Feedback	MQFB_NONE
Encoding	MQENC_NORMAL
CodedCharSetId	1208 (UTF-8)
Format	See Payload
Priority	Value taken from the `priority` field in the AMQP message header. If set, limited to a maximum of 9. If not set, takes the default value of 4.
Persistence	If the `durable` field in the AMQP message header is set true, set to MQPER_PERSISTENT. Otherwise, set to MQPER_NOT_PERSISTENT.
MagId	The queue manager allocates a unique 24-byte MsgId.
CorrelId	Value taken from the `correlation-id` field in the AMQP properties, if set. Set to a 24-byte binary value. Otherwise, set to MQCI_NONE/.

Table 103. Message descriptor for AMQP message (continued)	
Field	**Value**
BackoutCount	0
ReplyToQ	Value taken from the `reply-to` field in the AMQP properties, if set. Otherwise set to "".
ReplyToQMgr	""
Report	Value derived from any JMS IBM Report properties set in the AMQP application properties.
UserIdentifier	Set to the identifier of the authenticated user that connected to the AMQP channel
AccountingToken	MQACT_NONE
ApplIdentityData	Hexadecimal string. Set to the last 8 bytes of the MQ connection identifier of the AMQP channel.
PutApplType	MQAT_AMQP
PutApplName	
PutDate	Value taken from the `creation-time` field of the AMQP properties, if set. Otherwise set to the current date.
PutTime	Value taken from the `creation-time` field of the AMQP properties, if set. Otherwise set to the current time.
ApplOriginData	""

Message properties

There are two reasons for setting message properties:

- To allow parts of the AMQP message to flow through the queue manager without affecting the payload of the message.
- To allow selection of the `application-properties`.

The following table shows the properties that are set from the AMQP message:

Table 104. AMQP message properties			
Property name	**MQRFH2 name**	**Type**	**Description**
AMQPListener	mq_amqp.Lis	MQTYPE_STRING	An identifying string for the AMQP channel. It is used to generate the message, so that interested parties can tell which version put the message (for example, the service team when diagnosing problems). The value is not validated by the queue manager, and must not be documented externally.
AMQPVersion	mq_amqp.Ver	MQTYPE_STRING	The version of the AMQP message. If not present, "1.0" is assumed. The value is not validated by the queue manager.

Table 104. AMQP message properties (continued)

Property name	MQRFH2 name	Type	Description
AMQPClient	mq_amqp.Cli	MQTYPE_STRING	An identifying string for the API. It is used to send the AMQP message to the channel, so that interested parties can tell which version put the message (for example, the service team when diagnosing problems). The value is not validated by the queue manager and must not be documented externally.
AMQPDurable	mq_amqp.Hdr.Dur	MQTYPE_BOOLEAN	The value of the `durable` field in the AMQP message header, if set.
AMQPPriority	mq_amqp.Hdr.Pri	MQTYPE_INT32	The value of the `priority` field in the AMQP message header, if set.
AMQPTtl	mq_amqp.Hdr.Ttl	MQTYPE_INT64	The value of the `ttl` field in the AMQP message header, if set.
AMQPFirstAcquirer	mq_amqp.Hdr.Fac	MQTYPE_BOOLEAN	The value of the `first-acquirer` field in the AMQP message header, if set.
AMQPDeliveryCount	mq_amqp.Hdr.Dct	MQTYPE_INT64	The value of the `delivery-count` field in the AMQP message header, if set.
AMQPMsgId	mq_amqp.Prp.Mid	MQTYPE_STRING	The value of the `message-id` field in the AMQP properties, if set as a string.
		MQTYPE_BYTE_STRING	The value of the `message-id` field in the AMQP properties, if set as a byte string.
AMQPUserId	mq_amqp.Prp.Uid	MQTYPE_BYTE_STRING	The value of the `user-id` field in the AMQP properties, if set.
AMQPTo	mq_amqp.Prp.To	MQTYPE_STRING	The value of the `to` field in the AMQP properties, if set.
AMQPSubject	mq_amqp.Prp.Sub	MQTYPE_STRING	The value of the `subject` field in the AMQP properties, if set.
AMQPReplyTo	mq_amqp.Prp.Rto	MQTYPE_STRING	The value of the `reply-to` field in the AMQP properties, if set.
AMQPCorrelationId	mq_amqp.Prp.Cid	MQTYPE_STRING	The value of the `correlation-id` field in the AMQP properties, if set as a string..
		MQTYPE_BYTE_STRING	The value of the `correlation-id` field in the AMQP properties, if set as a byte string.
AMQPContentType	mq_amqp.Prp.Cnt	MQTYPE_STRING	The value of the `content-type` field in the AMQP properties, if set.

Table 104. AMQP message properties (continued)			
Property name	**MQRFH2 name**	**Type**	**Description**
AMQPContentEncoding	mq_amqp.Prp.Cne	MQTYPE_STRING	The value of the `content-encoding` field in the AMQP properties, if set.
AMQPAbsoluteExpiryTime	mq_amqp.Prp.Aet	MQTYPE_STRING	The value of the `absolute-expiry-time` field in the AMQP properties, if set.
AMQPCreationTime	mq_amqp.Prp.Crt	MQTYPE_STRING	The value of the `creation-time` field in the AMQP properties, if set.
AMQPGroupId	mq_amqp.Prp.Gid	MQTYPE_STRING	The value of the `group-id` field in the AMQP properties, if set.
AMQPGroupSequence	mq_amqp.Prp.Gsq	MQTYPE_INT64	The value of the `group-sequence` field in the AMQP properties, if set.
AMQPReplyToGroupId	mq_amqp.Prp.Rtg	MQTYPE_STRING	The value of the `reply-to-group-id` field in the AMQP properties, if set.

Each of the application-properties from the AMQP message is set as an IBM MQ message property. The `application-properties` section must be reconstituted identically byte-for-byte, and so the following restrictions apply:

- If an application property is rejected by the MQSETMP validation code, the message to be rejected. For example:
  - The property name is limited in length to MQ_MAX_PROPERTY_NAME_LENGTH.
  - The property name must follow the rules defined by the Java Language Specification for Java Identifiers.
  - The property name must not begin JMS or `usr.`JMS except for the documented JMS properties that can be set.
  - The property name must not be a SQL keyword.
- An application property containing Unicode character U+002E (".") causes the message to be rejected. The property must be expressible in the "usr" group of properties used by JMS.
- Only null, boolean, byte, short, int, long, float, double, binary and string properties are supported. An application property with any other type will cause the message to be rejected.

You can set the following JMS properties using `application-properties`:

- JMS_IBM_REPORT_EXCEPTION
- JMS_IBM_REPORT_EXPIRATION
- JMS_IBM_REPORT_COA
- JMS_IBM_REPORT_COD
- JMS_IBM_REPORT_PAN
- JMS_IBM_REPORT_NAN
- JMS_IBM_REPORT_PASS_MSG_ID
- JMS_IBM_REPORT_PASS_CORREL_ID
- JMS_IBM_REPORT_DISCARD_MSG

Note that property names and values are consistent with the equivalent “Mapping JMS provider-specific fields” on page 153 details, and that values that are not valid are ignored.

payload

- For an AMQP body with a single binary data section, the binary data (excluding the AMQP bits) is put as the IBM MQ payload, with a Format of MQFMT_NONE.
- For an AMQP body with a single string data section, the string data (excluding the AMQP bits) is put as the IBM MQ payload, with a Format of MQFMT_STRING.
- Otherwise, the AMQP body forms the payload as-is, with a Format of MQFMT_AMQP.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW Message delivery reliability

This section compares the reliability features for MQ Light API and Apache Qpid JMS.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW MQ Light message reliability

There are four features of the MQ Light API that allow you to control the reliability of message delivery to, and from, AMQP applications.

These are:

- “Message quality of service (QOS)” on page 667
- “Subscriber auto-confirm” on page 668
- “Subscription time to live” on page 668
- “Message persistence” on page 668

Message quality of service (QOS)

The MQ Light API offers two qualities of service:

- At most once
- At least once

You can choose which quality of service you want publishers and subscribers to use.

If you are using an MQ Light client, set the client or subscribe **qos** option to *QOS_AT_MOST_ONCE* or *QOS_AT_LEAST_ONCE*.

If you are using a different AMQP client, set the **settled** attribute of the transfer frame (for publishers), or the disposition frame (for subscribers) to *true* or *false*, depending on the quality of service you want to achieve.

The quality of service determines when a message is discarded from the `sending` side of a conversation:

Publishing

- If a publisher chooses **QOS** *0* (at most once) the publisher does not wait for an acknowledgment from the queue manager, before it discards its copy of the message. If the connection to the queue manager fails before the send has completed, the message might not be received by subscribers.
- If a publisher chooses **QOS** *1* (at least once) the publisher waits for the queue manager to acknowledge that the message has been written to subscriber queues before discarding its copy of the message. If the connection to the queue manager fails during the send, the publisher re-sends the message once it has reconnected to the queue manager.

Subscribing

- If a subscriber chooses **QOS** *0* the queue manager does not wait for an acknowledgment from the subscriber before discarding its copy of the message. If the connection to the subscriber fails before the subscriber has received the message, that message might be lost.
- If a subscriber chooses **QOS** *1* the queue manager waits for an acknowledgment from the subscriber before discarding its copy of the message. V 9.4.0 From IBM MQ 9.3.3, acknowledged messages are removed in batches to improve the performance. For more information, see "Removing acknowledged AMQP messages from the queue in batches" on page 670.

  If the connection to the subscriber fails before the subscriber has received the message, the message is kept by the queue manager. The queue manager re-sends the message to the subscriber when the queue manager reconnects, or to another subscriber if the subscription is shared.

Subscriber auto-confirm

If a subscriber chooses **QOS** *1* (at least once) it must acknowledge receipt of each message before the queue manager discards its copy. The subscriber can decide when it acknowledges messages.

With **auto-confirm** set to *true*, the MQ Light client automatically acknowledges delivery of each message, once the client has successfully received the message over the network.

This ensures that if there is a network failure, the message is redelivered to the application. However, it is still possible for the application to lose the message, if the application fails in between the MQ Light client acknowledging the message, and the application processing it.

With **auto-confirm** set to *false*, the MQ Light client does not automatically acknowledge delivery of the message, but leaves it to the application to decide when it should be acknowledged.

This allows an application to make an update to an external resource, such as a database or a file, before acknowledging to the queue manager that the message has now been processed and can be discarded.

Subscription time to live

When an application subscribes, it chooses whether the subscription, and the destination where messages are stored for that subscription, continue to exist after the application disconnects.

The MQ Light subscribe option **ttl** is used to specify the time (in milliseconds) that a subscription continues to exist after the application disconnects. If the application reconnects before this time, the subscription is resumed and the application can continue to consume messages from that subscription.

If the time-to-live period passes without the application reconnecting, the subscription is removed and any messages stored on its destination are lost, even if they are persistent messages.

If it is important not to lose messages, you must specify a time-to-live value to the application, that is high enough to ensure messages are not lost during an outage.

Message persistence

The persistence of messages is controlled by the publishing and subscribing applications, and the configuration of IBM MQ topic objects.

If the AMQP subscriber uses **QOS** *0* (at most once) and creates a non-durable subscription, the AMQP channel always puts non-persistent messages to the subscriber queue, regardless of the other options described in the following text.

Note that, if the queue manager is stopped both the subscription and the messages will be lost.

If an AMQP publisher sets the AMQP **durable** header to *true*, the AMQP channel puts persistent messages to subscriber queues.

If the queue manager is stopped for any reason, the messages are still available to subscribers when the queue manager is restarted.

If the **durable** header is not set, the AMQP channel chooses the persistence of published messages based on the **DEFPSIST** attribute of the relevant IBM MQ topic object.

By default, this is the SYSTEM.BASE.TOPIC, which uses a **DEFPSIST** attribute of *NO* (non-persistent).

Attention: Later versions of the MQ Light client do not support setting the AMQP durable header.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW Apache Qpid JMS message reliability

There are four features of the Apache Qpid™ JMS library that allow you to control the reliability of message delivery to, and from, AMQP applications.

These are for:

- "Publishing" on page 669/Producer for point to point messaging
  - Message expiration
  - Message persistence
- "Subscribing" on page 669
  - Subscription durability
  - Session acknowledgment mode (Applicable also for consumer point to point messaging)

Publishing

Message expiration

Setting the time-to-live value of the JMS producer affects the expiry time given to messages published by that message producer.

Ensure that the time-to-live value for a JMS producer is sufficiently large that messages are consumed before they expire.

Alternatively, leaving the time-to-live value unset prevents the message from expiring from the subscription queue.

Message persistence

Setting the delivery mode of the JMS message producer sets the persistence of the IBM MQ message published to the specified topic.

Ensure that you use **DeliveryMode**.PERSISTENT for messages that must be retained when a queue manager is ended, or has an outage.

Subscribing

Subscription durability

AMQP channels support the creation of durable subscriptions by using the durable versions of the JMS create consumer methods:

- **createDurableConsumer()**
- **createSharedDurableConsumer()**

Session acknowledgment mode

To guarantee that a consumed message has been fully processed before it is removed from the IBM MQ subscription queue, create a JMS session using the **Session**.CLIENT_ACKNOWLEDGE mode and use the **message.acknowledge()** method to acknowledge this message and any others previously received on this session.

Related concepts

Developing AMQP client applications

The IBM MQ support for AMQP APIs, allows an IBM MQ administrator to create an AMQP channel. When it is started, this channel defines a port number that accepts connections from AMQP client applications.

V 9.4.0 Removing acknowledged AMQP messages from the queue in batches

If an AMQP application is using QOS_AT_LEAST_ONCE(1) message delivery, the AMQP service waits for an acknowledgment from the application before it discards the copy of a message that it keeps after it sends that message to the application. From IBM MQ 9.3.3, messages that have been acknowledged are removed from the queue in batches, instead of individually, resulting in improved performance.

About this task

Before IBM MQ 9.4.0, each message is removed from the queue individually.

From IBM MQ 9.4.0, you can use the two system properties **com.ibm.mq.AMQP.BATCHSZ** and **com.ibm.mq.AMQP.BATCHINT** to fine tune the processing of acknowledgments in batches for improved performance:

com.ibm.mq.AMQP.BATCHSZ
: This attribute defines the maximum number of acknowledgments to be received before the AMQP service removes messages. The number can be in the range 1 through 9999. If an invalid number is set, or if the specified number is out of range, the default value of 50 is used.

  The batch size does not affect the way that the messages are transferred. Messages are always transferred individually, but are then removed in a batch after the AMQP service receives the acknowledgments. The actual size of a batch can be less than the value specified by **com.ibm.mq.AMQP.BATCHINT**. For example, a batch completes if the period set by the **com.ibm.mq.AMQP.BATCHINT** attribute expires.

com.ibm.mq.AMQP.BATCHINT
: This attribute defines the amount of time, in milliseconds, for which the AMQP service keeps acknowledged messages on the queue. If the batch is not full, then the batch is cleared after this duration. You can specify any number of milliseconds, from 1 through 999 999 999. The default value is 50. If you do not specify a value for this attribute, the default value of 50 is used.

Notes:

1. Whether the AMQP service waits for an acknowledgment before it discards a message depends on which of the following two qualities of service an application is using for message delivery:

   - QOS for QOS_AT_MOST_ONCE(0)

     If an AMQP application is using this quality of service, it does not acknowledge messages, so the AMQP service discards messages after it sends them to the application without waiting for an acknowledgment.

   - QOS_AT_LEAST_ONCE(1)

     If an AMQP application is using this quality of service, it does acknowledge messages, so the AMQP service keeps a copy of each message after it sends it to the application until it receives an acknowledgment from the application. If the application disconnects from or loses the connection before acknowledging the message, the message is made available to other applications. The AMQP service does not remove a message from the queue until it has been acknowledged.

2. MQ Appliance The **com.ibm.mq.AMQP.BATCHSZ** and **com.ibm.mq.AMQP.BATCHINT** system properties are not applicable on IBM MQ Appliance. A default value of 50 is used on IBM MQ Appliance.

.

Procedure

Use the **com.ibm.mq.AMQP.BATCHSZ** and **com.ibm.mq.AMQP.BATCHINT** system properties to fine tune the processing of acknowledgments in batches.

From IBM MQ 9.3.3, when the queue manager is created, the `amqp_java.properties` file contains the following default values for the system properties:

```
-Dcom.ibm.mq.AMQP.BATCHSIZE=50
-Dcom.ibm.mq.AMQP.BATCHINT=50
```

Depending on the message rate consumed, you can fine tune processing of acknowledgments in batches for improved performance. A migrated queue manager does not have these properties in the `amqp_java.properties` file. So, for a migrated queue manager, or if the properties are not set, the default values are used. You can add these properties to fine tune the values for optimized performance.

Acknowledged messages are removed in batches when one of the following conditions is met:

- The number of acknowledged messages reaches **com.ibm.mq.AMQP.BATCHSZ**.
- **com.ibm.mq.AMQP.BATCHINT** is exceeded after the start of the batch.
- The application disconnects or closes the queue or topic before the two previous conditions are satisfied.

ALW Topologies for AMQP clients with IBM MQ

Example topologies to help you develop your AMQP clients to work with IBM MQ.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW AMQP clients communicating over IBM MQ

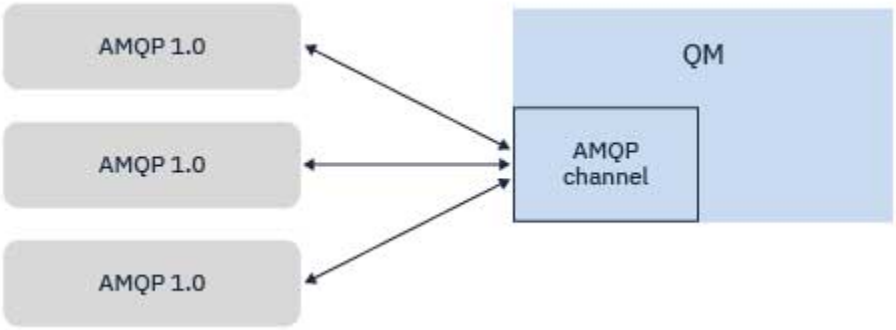
You can use IBM MQ as the messaging provider for any application that complies with AMQP 1.0. Although any AMQP 1.0 client can connect to an AMQP channel, some AMQP features are not supported, for example transactions or multiple sessions.

By defining one or more AMQP channels, AMQP 1.0 clients can connect to the queue manager and send messages to a topic string. Clients can also subscribe to a topic pattern to receive messages that match the pattern.

In the following scenario, the only applications sending and receiving messages are AMQP 1.0 applications.

Applications can choose whether the destinations created by subscribing to a topic string are persistent so that messages are not lost if the application temporarily loses its connection to the queue manager.

Applications can also choose how long messages are kept before being purged from the destination.



Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW AMQP clients exchanging messages with IBM MQ applications

By defining and starting an AMQP channel, AMQP 1.0 applications can publish messages that are received by existing MQ applications. Messages published through an AMQP channel are all sent to MQ topics, not MQ queues. An MQ application that has created a subscription using the MQSUB API call receives messages published by AMQP 1.0 applications, provided that the topic string or topic object used by the MQ application matches the topic string published by the AMQP client.

AMQP message data, attributes, and properties are set on the MQ message received by the MQ application. For more information about AMQP to MQ message mappings, see “Mapping AMQP fields onto IBM MQ fields (incoming messages)” on page 663.

If the MQ application has created a subscription that is durable, messages published by the AMQP application are stored on the queue that backs the subscription. The messages are then received by the MQ application when the application resumes its subscription. If the AMQP application specifies a message time to live and the MQ application does not reconnect within the time to live, the message is expired from the queue.

AMQP 1.0 applications can also consume messages that are published by existing MQ applications. Messages published by MQ applications to an MQ topic or topic string are received by an AMQP 1.0 application provided that the application has subscribed with a topic pattern that matches the published topic string.

If the AMQP 1.0 application specifies a time-to-live value for the subscription, and the AMQP application disconnects for greater than the time to live, the subscription is expired from the queue manager and any messages stored on the subscription queue are lost.

MQMD fields, message properties, and application data are set on the AMQP message received by the AMQP application. For more information about the MQ to AMQP message mappings, see “Mapping IBM MQ fields onto AMQP fields (outgoing messages)” on page 661.

Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW Configuring AMQP clients to interact directly with applications on IBM MQ queues

The IBM MQ AMQP implementation supports publish/subscribe and point to point. For any AMQP client that does not support point to point use the following steps to send messages to a queue, or receive messages from a queue.

Overview

For example, assume there is an application getting messages from an input queue IN_QUEUE and putting those messages to an output queue OUT_QUEUE. It is possible for AMQP clients to put messages to IN_QUEUE, and get messages from OUT_QUEUE

Note: There are no changes required to the application itself.

For an AMQP publisher to put a message to a queue, you need to create an administrative subscription for the topic string the AMQP client is publishing to, with a destination of the intended queue; see “Sending messages to the application:” on page 673.

For an AMQP subscriber to get messages from a queue, you need to replace the queue with an alias queue of the same name, with a target of a topic object representing the topic string the AMQP client is subscribed to; see “Getting messages from the application:” on page 673

Sending messages to the application:

The application is already picking up messages from IN_QUEUE and you want an AMQP client to be able to publish messages, so that they go to this queue to be processed by the application.

To do this, you create a new administrative subscription, where the topic string this subscription receives messages from, is the topic string that the AMQP client publishes to. The destination queue of this subscription is the input queue for the application, IN_QUEUE.

Any messages that are published to the defined topic string for that administrative subscription are routed to the defined destination, in this case IN_QUEUE.

Assuming the AMQP client publishes to a topic string /application/in, you can create an administrative subscription APP_IN, using the following MQSC command:

```
DEF SUB(APP_IN) TOPICSTR('/application/in') DEST('IN_QUEUE')
```

When you have defined this object, all messages published to /application/in are routed to the destination IN_QUEUE, where they are picked up by the application in the same way as any other messages put to this queue by other applications.

Getting messages from the application:

The application is putting messages to OUT_QUEUE, where they can be picked up and processed by other clients.

However, in this case, you want the messages to be delivered to an AMQP client instead, but AMQP clients only use publish/subscribe, and cannot pick messages up directly from a queue.

To replace the clients previously receiving message with the subscribing AMQP client, you need to create a topic object, for the topic string that the AMQP client is subscribed to, and an alias queue.

Attention: If you define the alias queue, and then start the producing application before any AMQP clients have had a chance to subscribe, messages the producing application sends to the "queue" (now a topic) will be lost because there are no subscribers.

The changes described in this text, replace the clients previously receiving messages with the subscribing AMQP client only. To use a combination of AMQP and other clients to get messages, more extensive changes are required.

Assuming the AMQP client subscribes to a topic string `/application/out`, you can define the topic object APP_OUT using the following MQSC command:

```
DEF TOPIC(APP_OUT) TOPICSTR('/application/out')
```

Any messages delivered to this topic object are delivered to the AMQP client subscribing to the same topic string.

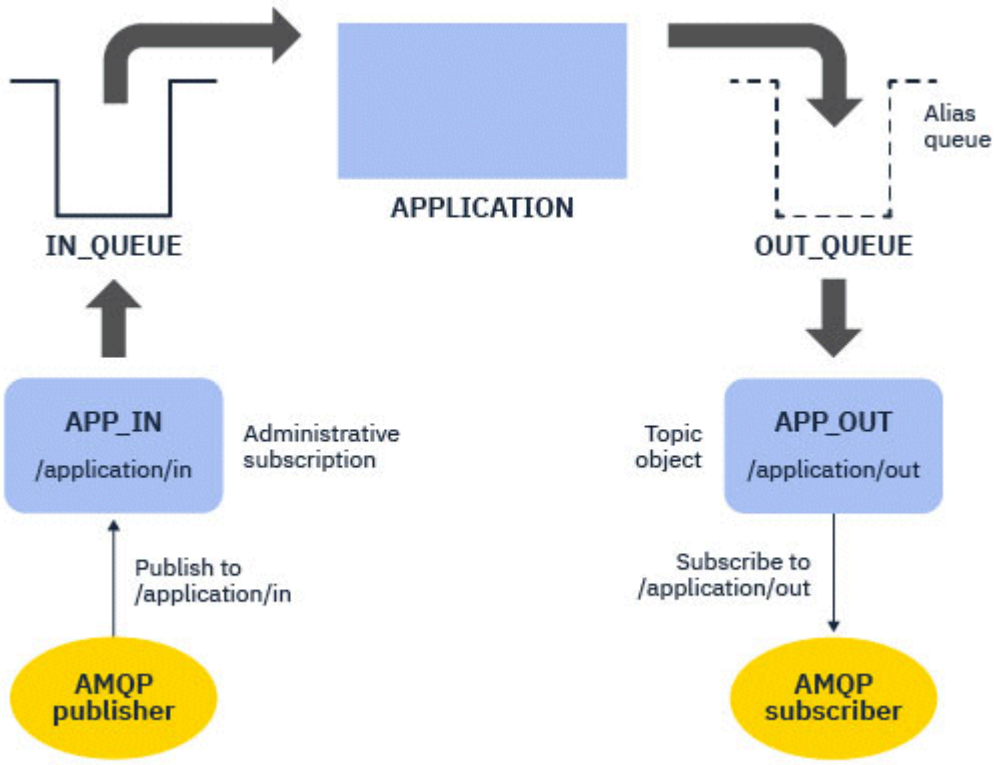
You then need to ensure that messages put to OUT_QUEUE by the application are delivered to this new topic object, so that they are sent to the subscribing client.

To do this, replace the existing queue OUT_QUEUE with an alias queue of the same name, with a target type of the topic object just created, using the following MQSC command:

```
DEF QALIAS(OUT_QUEUE) TARGTYPE(TOPIC) TARGET(APP_OUT)
```

Now, messages put by the application to OUT_QUEUE do not wait on the queue to be picked up; instead they are delivered to the target of this alias queue, that is, the new topic object APP_OUT.

The AMQP client, which is subscribed to the topic string represented by this topic object `/application/out`, then receives any messages sent to this topic object from the alias queue.



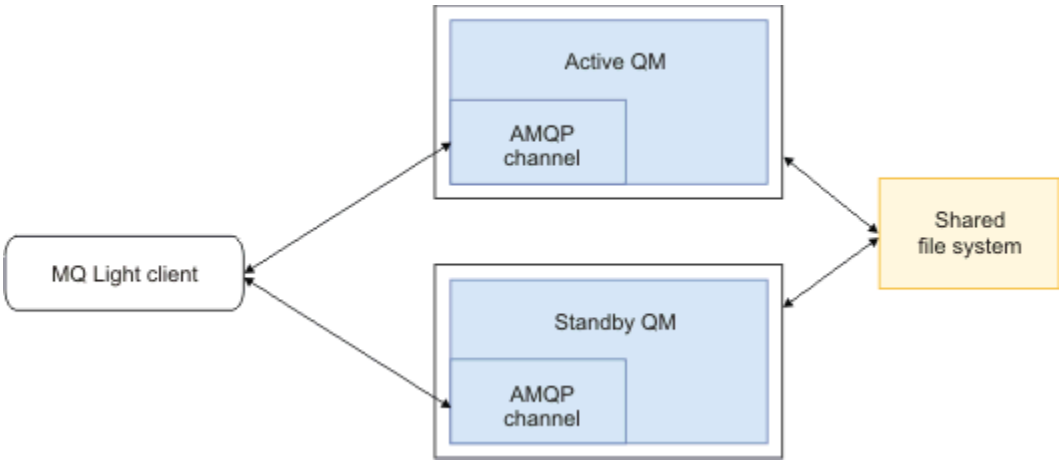
Related tasks

Creating and using AMQP channels

ALW Configuring an AMQP client for high availability

You can configure AMQP 1.0 applications to connect to the active instance of an IBM MQ multi-instance queue manager and fail over to the standby instance of the multi-instance queue manager in a high availability (HA) pair. To do this, you configure the AMQP application with two IP addresses and port pairs.

You can configure the AMQP client API with a custom function, which is called if the client loses its connection to the server. The function can connect to an alternative IP address, for example a standby IBM MQ queue manager or to the original IP address. For other AMQP clients, if the client supports configuration of multiple connection endpoints, configure the application with two host-port pairs and use the reconnect features provided by the AMQP library to switch to the standby queue manager.



Related tasks

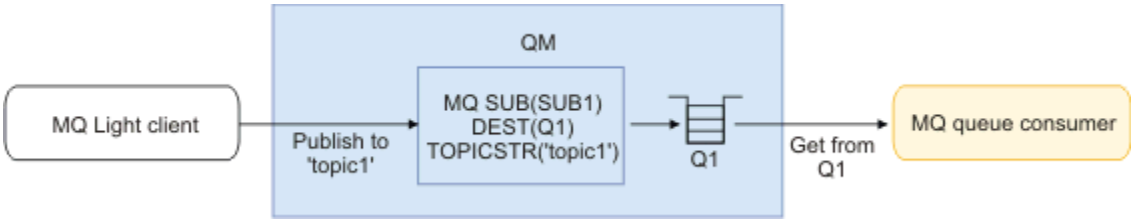
Creating and using AMQP channels

Securing AMQP clients

ALW Configuring publish/subscribe for AMQP clients

AMQP clients can publish to a topic with an IBM MQ subscription that routes messages to an IBM MQ queue read by an existing application. If you want an AMQP 1.0 application to send messages to an existing IBM MQ application that is configured to read from a queue, you must define an administered IBM MQ subscription on the queue manager.

Configure the subscription to use a topic pattern that matches the topic string used by the AMQP application. Set the subscription destination to the name of the queue that the IBM MQ application gets or browses messages from.



Related tasks

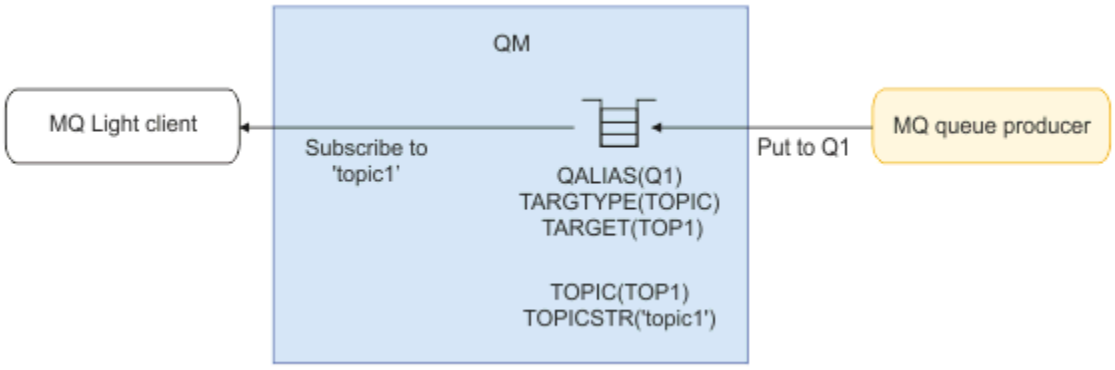
Creating and using AMQP channels

Securing AMQP clients

ALW AMQP client using a queue alias to receive messages from an IBM MQ application

An AMQP client can subscribe to a topic and receive messages put to an alias queue by an IBM MQ application. If you want an AMQP 1.0 application to receive messages from an existing IBM MQ application that is configured to put messages on a queue, you must define a queue alias (QALIAS) on the queue manager.

The queue alias must have the same name as the queue that the IBM MQ application opens for putting. The queue alias must specify a base type of TOPIC and a base object of an IBM MQ topic object that has a topic string that matches the topic pattern subscribed to by the AMQP application.



Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW AMQP client submitting requests to and consuming responses from an application server

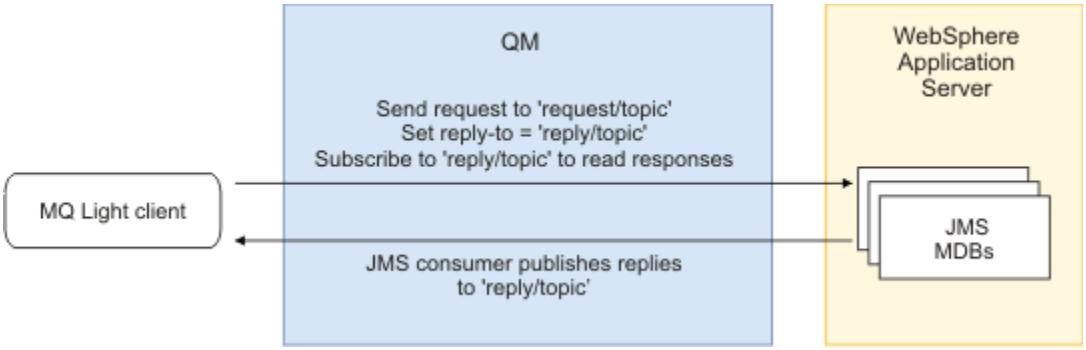
An AMQP client can submit requests to a message-driven bean running in an application server and consume responses from a reply topic. IBM MQ supports AMQP 1.0 applications setting a reply-to topic in the messages that IBM MQ publishes. When an AMQP message is published with the reply-to attribute set, the value of the reply-to field is set as a JMS property for JMS consumers to receive. This setting allows JMS consumers to read the reply-to topic from the message and send a response message back to the AMQP client.

The JMS property is **JMSReplyTo**. The AMQP reply-to string must be one of the following types:

- A topic string. For example, `'reply/topic'`
- An AMQP address URL in the form `amqp://host:port/[topic-string]`. For example, `amqp://localhost:5672/reply/topic`

If you specify an AMQP address URL as the reply-to field, everything except the topic-string at the end of the URL is removed before setting the **JMSReplyTo** property.

For more information about the mappings from an AMQP reply-to address to a **JMSReplyTo** property, see “Mapping AMQP fields onto IBM MQ fields (incoming messages)” on page 663



Related tasks

Creating and using AMQP channels

Securing AMQP clients

ALW Interoperability between MQ Light and Apache Qpid JMS applications

MQ Light and Apache Qpid JMS applications work in similar ways, and when subscribing to a topic, create IBM MQ subscriptions that follow the same naming convention.

Private, non-shared subscription

The name of the IBM MQ subscription created by the application is `:private:<clientid>:<topicstring>`.

An application using a different client ID is not able to access the subscriptions created by a other applications, because the subscription name is automatically generated and includes the AMQP client ID.

Both Apache Qpid JMS and MQ Light applications use this naming convention for private subscriptions.

Globally shared subscriptions

The name of a globally shared IBM MQ subscription created by an AMQP client is `:share:<sharename>:<topicstring>`.

If several applications with different AMQP client IDs specify the same share name and topic string, they share a single subscription, and can work together to process the messages for that subscription. You can use this pattern if you want to scale the number of worker applications draining messages from a subscription.

Both Apache Qpid JMS and MQ Light applications use this naming convention for globally shared subscriptions. In the case of Apache Qpid JMS, this requires that the JMS connection does not have a client ID specified on it.

The Apache Qpid JMS library generates an AMQP client ID automatically, but this client ID is not used for the purposes of IBM MQ subscription naming.

Note: Globally shared subscriptions are still scoped to an individual queue manager.

Private shared subscriptions

The name of a privately shared IBM MQ subscription created by an AMQP client is `:privateshare:<clientid>:<sharename>:<topicstring>`.

If several threads from a single Apache Qpid JMS application use the same share name and topic string, and a client ID has been configured on the JMS connection, those threads share the same IBM MQ subscription object.

However, other Apache Qpid JMS connections are not able to share the subscription because they must use a different client ID.

MQ Light clients do not support the concept of private shared subscriptions, and cannot consume messages from a private shared subscription created by an Apache Qpid JMS application.

IBM MQ JMS subscriptions

IBM MQ JMS subscriptions use a different naming scheme from AMQP channels. It is not possible for MQ Light or Apache Qpid JMS applications to share subscriptions with IBM MQ JMS applications.

Related concepts

Developing AMQP client applications

The IBM MQ support for AMQP APIs, allows an IBM MQ administrator to create an AMQP channel. When it is started, this channel defines a port number that accepts connections from AMQP client applications.

ALW IBM MQ AMQP listener control properties

For better performance in a multi-threaded application, you can tune the number of worker threads that the AMQP service should use by configuring a property in the AMQP properties file.

You can configure AMQP listener service properties in the following properties files:

- Windows On Windows systems: `amqp_win.properties` .
- Linux AIX On AIX and Linux systems: `amqp_unix.properties` .

The properties that you can configure are as follows:

Table 105. AMQP listener service properties

Property	Description
com.ibm.mq.MQXR.Workers	The number of server worker threads that the AMQP listener service creates. If this value is not specified, it is defaulted to be equal to the number of logical processors on the system.
MQIBindType	The binding type for the AMQP service, either: FASTPATH, SHARED, or ISOLATED. The default is FASTPATH.

The AMQP listener service balances the client connection workload across a number of worker threads. The number of worker threads that the AMQP service should use can be specified using the **com.ibm.mq.MQXR.Workers** property.

The IBM MQ queue manager administrator can tune the number of worker threads for better performance in a multi-threaded application. Typically, best performance is achieved when the number of worker threads matches the number of logical processors on the system. However, this might not always be the case for certain machine configurations and client load characteristics, so an element of tuning might be required to find the optimal value for the number of worker threads.

Before tuning, make sure that you thoroughly understand the nature of your client applications and their workloads. Measuring the performance of your application with different thread counts and benchmarking should help to determine the optimal value for the number of worker threads.

Note: MQ Appliance These properties are not applicable on IBM MQ Appliance. Default values are used on IBM MQ Appliance.

Developing REST applications with IBM MQ

You can develop REST applications to send and receive messages. IBM MQ supports different REST APIs depending upon platform and capability.

The following options are the IBM MQ supported options you can choose from to send messages to, and receive messages from, IBM MQ:

- IBM MQ messaging REST API
- IBM z/OS Connect EE
- IBM Integration Bus
- DataPower

IBM MQ messaging REST API

You can use the messaging REST API to send, receive, and browse IBM MQ messages in plain text format. The messaging REST API is enabled by default.

Support is provided for a number of different HTTP headers which can be used to set common message properties.

The messaging REST API is fully integrated with IBM MQ security. To use the messaging REST API, users must be authenticated to the mqweb server and must be a member of the MQWebUser role.

For further information, see “Messaging using the REST API” on page 680. See also Tutorial: Get started with the IBM MQ messaging REST API on IBM Developer, which includes Go and Node.js examples for using the messaging REST API.

IBM z/OS Connect EE

IBM z/OS Connect EE is a z/OS product that allows you to build REST APIs on top of existing z/OS assets, such as CICS or IMS transactions, and IBM MQ queues and topics. The existing z/OS asset is hidden from the user. This allows you to REST enable your assets without changing them or any of the existing applications that use them.

IBM z/OS Connect EE provides automatic data transformation to translate between the JSON data used by the REST APIs and the more traditional language structures, for example COBOL, expected by many mainframe applications.

The Eclipse based IBM z/OS Connect EE API toolkit can be used to build up a comprehensive RESTful API making use of query parameters and URL path segments, manipulating the JSON format as it flows through the IBM z/OS Connect EE runtime.

IBM z/OS Connect EE can be used to expose IBM MQ queues and topics as RESTful APIs through the IBM MQ service provider. Two different service types are supported:

- One-way services: these provide a REST API that allows a single IBM MQ operation to be performed on a queue or topic. Depending on the exact configuration an HTTP request can result in a message being sent to a queue or published to a topic; or an HTTP request can result in a message being destructively received from a queue
- Two-way services: these provide a REST API on top of a pair of queues used by a back-end request-response style application. Callers issue an HTTP request to the two-way service. The HTTP request payload is transformed from JSON to a traditional language structure and put onto a request queue where it is processed by the back-end application and a response placed on the response queue. This response is retrieved by the service, converted from the traditional language structure to JSON and sent back to the caller as the POST response body.

For more information on IBM z/OS Connect EE, see z/OS Connect EE.

For more information on the IBM MQ service provider, see Using the IBM MQ service provider.

IBM Integration Bus

IBM Integration Bus is IBM's leading integration technology which can be used to connect applications and systems together regardless of the message formats and protocols that they support.

IBM Integration Bus has always supported IBM MQ and provides *HTTPInput* and *HTTPRequest* nodes which can be used to construct a RESTful interface on top of IBM MQ, and many other systems such as databases.

IBM Integration Bus can be used to do much more than provide a simple REST interface on top of IBM MQ. Its capabilities can be used to provide advanced payload manipulation, payload enrichment, and many other enhancements as part of a REST API.

For further information, see the technology sample which exposes a JSON over REST interface on top of an IBM MQ application that expects an XML payload.

DataPower

The DataPower gateway is a single multi-channel gateway which helps provide security, control, integration and optimized access to a range of systems, including IBM MQ. It comes in both hardware and virtual form factors.

One of the services that DataPower provides is a multi-protocol gateway which can take input in one protocol and generate output in a different protocol. In particular DataPower can be configured to accept HTTP(S) data and route it to IBM MQ over a client connection, which can be used to build a REST interface on top of IBM MQ. Other DataPower services such as transformation can also be used to enhance the REST interface.

For further information, see Multi-Protocol Gateway.

Messaging using the REST API

You can use the messaging REST API to perform simple point-to-point and publish messaging. You can publish messages to a topic, send messages to a queue, browse messages on a queue, and destructively get messages from a queue. Information is sent to, and received from, the messaging REST API in plain text format.

Before you begin

Note:

- The messaging REST API is enabled by default. You can disable the messaging REST API to prevent all messaging. For more information about enabling or disabling the messaging REST API, see Configuring the messaging REST API.
- The messaging REST API is integrated with IBM MQ security. To use the messaging REST API, users must be authenticated to the mqweb server and must be a member of the MQWebUser role. The user must also be authorized to access the specified queue or topic. For more information about security for the REST API, see IBM MQ Console and REST API security.
- If you use Advanced Message Security (AMS) with the messaging REST API, note that all messages are encrypted by using the context of the mqweb server, not the context of the user that posts the message.
- When receiving or browsing a message, only IBM MQ `MQSTR` or `JMS TextMessage` formatted messages are supported. Subsequently, all messages are destructively received under sync-point and any unhandled messages are left on the queue. The IBM MQ queue can be configured to move these poison messages to an alternate destination. For further information, see “Handling poison messages in IBM MQ classes for JMS” on page 225.
- The messaging REST API does not give you once-and-once only delivery of messages with transactional support. If an HTTP POST is issued and the connection fails before an HTTP response is received by the client, the client can not immediately tell if the message was sent to the specified queue, or published to the specified topic. If an HTTP DELETE is issued and the connection fails before an HTTP response

is received by the client, then a message might have been destructively got from the queue and lost, as there is no way of rolling the destructive get back.

- From IBM MQ 9.3.0, newlines in incoming strings are no longer removed by the HTTP POST operation. REST applications that use earlier versions should not use newlines in messages that are sent or published using the REST API, as they will be lost.

Procedure

- "Getting started with the messaging REST API" on page 681
- "Using the messaging REST API" on page 683
- REST API error handling
- REST API discovery
- REST API national language support

Related reference

Messaging REST API reference

Related information

Tutorial: Get started with the IBM MQ messaging REST API

Getting started with the messaging REST API

Get started quickly with the messaging REST API and try out a few example commands by using cURL.

Before you begin

To get you started with using the messaging REST API, the examples in this task have the following requirements:

- The examples use cURL to send REST requests to put and get messages from a queue. Therefore, to complete this task you need cURL installed on your system.
- The examples use a queue manager `QM1`. Either create a queue manager with the same name, or substitute an existing queue manager on your system. The queue manager must be on the same machine as the mqweb server.
- To complete this task, you must be a user with certain privileges so that you can use the **dspmqweb** command:
  - z/OS On z/OS, you must have authority to run the **dspmqweb** command, and write access to the `mqwebuser.xml` file.
  - Multi On all other operating systems, you must be a privileged user.

  IBM i On IBM i, the commands should be running in QSHELL.

Procedure

1. Ensure that the mqweb server is configured for the messaging REST API:
   - Ensure that you configured the mqweb server for use by the administrative REST API, the administrative REST API for MFT, the messaging REST API, or IBM MQ Console.For more information about configuring the mqweb server with a basic registry, see Basic configuration for the mqweb server.
   - If the mqweb server is already configured, ensure that you added the appropriate users to enable messaging in step 5 of Basic configuration for the mqweb server.
     - If **mqRestMessagingAdoptWebUserContext** is set to `true` in the mqweb server configuration, users of the messaging REST API must be a member of the `MQWebUser` role. The `MQWebAdmin` and `MQWebAdminRO` roles are not applicable for the messaging REST API. The

users must also be authorized to access queues and topics that are used for messaging through OAM or RACF®.

– If **mqRestMessagingAdoptWebUserContext** is set to `false` in the mqweb server configuration, the user ID that is used to start the mqweb server must be authorized to access queues used for messaging through OAM or RACF.

2. z/OS
On z/OS, set the WLP_USER_DIR environment variable so that you can use the **dspmqweb** command. Set the variable to point to your mqweb server configuration by entering the following command:

```
export WLP_USER_DIR=WLP_user_directory
```

where *WLP_user_directory* is the name of the directory that is passed to `crtmqweb`. For example:

```
export WLP_USER_DIR=/var/mqm/web/installation1
```

For more information, see Creating the mqweb server.

3. Determine the REST API URL by entering the following command:

```
dspmqweb status
```

The examples in the following steps assume that your REST API URL is the default URL `https://localhost:9443/ibmmq/rest/v2/`. If your URL is different than the default, substitute your URL in the following steps.

4. Create a queue, `MSGQ`, on queue manager `QM1`. This queue is used for messaging. Use one of the following methods:

- Use a POST request on the `mqsc` resource of the administrative REST API, authenticating as the `mqadmin` user:

```
curl -k https://localhost:9443/ibmmq/rest/v2/admin/action/qmgr/QM1/mqsc -X POST -u
mqadmin:mqadmin -H "ibm-mq-rest-csrf-token: value" -H "Content-Type: application/
json" --data "{\"type\": \"runCommandJSON\",\"command\": \"define\", \"qualifier\":
\"qlocal\",\"name\": \"MSGQ\"}"
```

- Use MQSC commands:

z/OS On z/OS, use a 2CR source instead of the **runmqsc** command. For more information, see Sources from which you can issue MQSC and PCF commands on IBM MQ for z/OS.

a. Start **runmqsc** for the queue manager by entering the following command:

```
runmqsc QM1
```

b. Use the **DEFINE QLOCAL** MQSC command to create the queue:

```
DEFINE QLOCAL(MSGQ)
```

c. Exit **runmqsc** by entering the following command:

```
end
```

5. Grant authority for the user that you added to the `mqwebuser.xml` in step 5 of Basic configuration for the mqweb server to access the queue `MSGQ`. Substitute your user where `myuser` is used:

- z/OS On z/OS:

a. Grant your user access to the queue:

```
RDEFINE MQQUEUE hlq.MSGQ UACC(NONE)
PERMIT hlq.MSGQ CLASS(MQQUEUE) ID(MYUSER) ACCESS(UPDATE)
```

b. Grant the mqweb started task user ID access to set all context on the queue:

```
RDEFINE MQADMIN hlq.CONTEXT.MSGQ UACC(NONE)
PERMIT hlq.CONTEXT.MSGQ CLASS(MQADMIN) ID(mqwebStartedTaskID) ACCESS(CONTROL)
```

- Multi On all other operating systems, if your user is in the mqm group, authority is already granted. Otherwise, enter the following commands:

  a. Start **runmqsc** for the queue manager by entering the following command:

  ```
  runmqsc QM1
  ```

  b. Use the **SET AUTHREC** MQSC command to give your user browse, inquire, get and put authorities on the queue:

  ```
  SET AUTHREC PROFILE(MSGQ) OBJTYPE(QUEUE) +
  PRINCIPAL(myuser) AUTHADD(BROWSE, INQ, GET, PUT)
  ```

  c. Exit **runmqsc** by entering the following command:

  ```
  end
  ```

6. Put a message with the content `Hello World!` on the queue `MSGQ` on queue manager `QM1`, by using a POST request on the `message` resource. Substitute your user ID and password from the `mqwebuser.xml` for `myuser` and `mypassword`:

   Basic authentication is used, and an `ibm-mq-rest-csrf-token` HTTP header with an arbitrary value is set in the cURL REST request. This additional header is required for POST, PATCH, and DELETE requests.

   ```
   curl -k https://localhost:9443/ibmmq/rest/v2/messaging/qmgr/QM1/queue/MSGQ/message -X
   POST -u myuser:mypassword -H "ibm-mq-rest-csrf-token: value" -H "Content-Type: text/
   plain;charset=utf-8" --data "Hello World!"
   ```

7. Destructively get the message from queue `Hello World!` on the queue `MSGQ` on queue manager `QM1`, by using a DELETE request on the `message` resource. Substitute your user ID and password from the `mqwebuser.xml` for `myuser` and `mypassword`:

   ```
   curl -k https://localhost:9443/ibmmq/rest/v2/messaging/qmgr/QM1/queue/MSGQ/message -X DELETE
   -u myuser:mypassword -H "ibm-mq-rest-csrf-token: value"
   ```

   The message `Hello World!` is returned.

What to do next

- The examples use basic authentication to secure the request. You can use token-based authentication or client-based authentication instead. For more information, see Using client certificate authentication with the REST API and IBM MQ Console, and Using token-based authentication with the REST API.
- Learn more about using the messaging REST API and constructing URLs with query parameters: “Using the messaging REST API” on page 683.
- When you use the messaging REST API, connections to the queue manager are pooled to optimize performance. You can configure the maximum pool size, and what action is taken when all the connections in the pool are in use: Configuring the messaging REST API.
- Browse the reference information for the available messaging REST API resources and all the available query parameters: messaging REST API reference.
- Discover the administrative REST API, a RESTful interface for IBM MQ administration: Administration using the REST API.
- Discover the IBM MQ Console, a browser-based GUI: Administration using the IBM MQ Console.

Using the messaging REST API

When you use the messaging REST API, you invoke HTTP methods on URLs to send and receive IBM MQ messages. The HTTP method, for example POST, represents the type of action to be performed on the

object that is represented by the URL. Further information about the action might be encoded in query parameters. Information about the result of performing the action might be returned as the body of the HTTP response.

Before you begin

Consider these things before you use the messaging REST API:

- You must authenticate with the mqweb server in order to use the messaging REST API. You can authenticate by using HTTP basic authentication, client certificate authentication, or token based authentication. For more information about how to use these authentication methods, see IBM MQ Console and REST API security.
- The REST API is case-sensitive. For example, an HTTP POST on the following URL results in an error if the queue manager is called `qmgr1`.

  ```
  /ibmmq/rest/v2/messaging/qmgr/QMGR1/queue/Q1/message
  ```

- V 9.4.0 If you are connecting to a remote queue manager with the messaging REST API, you must use the unique name for the queue manager connection instead of the queue manager name.
- Not all of the characters that can be used in IBM MQ object names can be directly encoded in a URL. To encode these characters correctly, you must use the appropriate URL encoding:
  - A forward slash must be encoded as %2F.
  - A percent sign must be encoded as %25.
  - A period must be encoded as %2E.
  - A question mark must be encoded as %3F.
- When receiving or browsing a message, only IBM MQ `MQSTR` and `JMS TextMessage` formatted messages are supported. Subsequently, all messages are destructively received under sync-point and any unhandled messages are left on the queue. The IBM MQ queue can be configured to move these poison messages to an alternate destination. For further information, see "Handling poison messages in IBM MQ classes for JMS" on page 225.

About this task

When you use the REST API to perform a messaging action on an IBM MQ queue object, you first need to construct a URL to represent that object. Each URL starts with a prefix, which describes which host name and port to send the request to. The rest of the URL describes a particular object, or route to that object, known as a resource.

The messaging action that is to be performed on the resource defines whether the URL needs query parameters or not. It also defines the HTTP method that is used, and whether additional information is sent to the URL, or returned from it. The additional information might form part of the HTTP request, or be returned as part of the HTTP response.

After you construct the URL, you can send the HTTP request to IBM MQ. You can send the request by using the HTTP implementation that is built into the programming language of your choice. You can also send the request by using command line tools such as cURL, or a web browser, or web browser add-on.

Important: You must, as a minimum, carry out steps "1.a" on page 684 and "1.b" on page 684.

Procedure

1. Construct the URL:
   a) Determine the prefix URL by entering the following command:

   ```
   dspmqweb status
   ```

   The URL that you want to use includes the `/ibmmq/rest/` phrase.
   b) Add the queue and associated queue manager resources to use for messaging to the URL path.

In the messaging reference, the variable segments can be identified in the URL by the braces that surround it *{ }*. For further information, see /messaging/qmgr/{qmgrName}/queue/{queueName}/message.

For example, to interact with queue *Q1* associated with queue manager *QM1*, add `/qmgr` and `/queue` to the prefix URL to create the following URL:

```
https://localhost:9443/ibmmq/rest/v2/messaging/qmgr/QM1/queue/Q1/message
```

Tip: V 9.4.0 If the queue manager is a remote queue manager, you must use the unique name for the queue manager in place of the queue manager name. The remote queue manager must be configured before it can be used with the messaging REST API. For more information, see "Setting up a remote queue manager to use with the messaging REST API" on page 685.

c) Optional: Add an optional query parameter to the URL.

Add a question mark, ?, query parameter, equal sign =, and a value to the URL.

For example, to wait for a maximum of 30 seconds for the next message to become available, create the following URL:

```
https://localhost:9443/ibmmq/rest/v2/messaging/qmgr/QM1/queue/Q1/message?wait=30000
```

d) Optional: Add further optional query parameters to the URL.

Add an ampersand, &, to the URL, and then repeat step 1c.

2. Invoke the relevant HTTP method on the URL. Specify any optional message payload, and provide the appropriate security credentials to authenticate. For example:
   - Use the HTTP/REST implementation of your chosen programming language.
   - Use a tool such as a REST client browser add-on or cURL.

V 9.4.0 Setting up a remote queue manager to use with the messaging REST API

You can use the messaging REST API to connect to remote queue managers for messaging. Before you can connect to a remote queue manager, you must set up the remote queue manager configuration. Then, you can connect to the remote queue manager by using the unique name that is defined in the configuration information.

Before you begin

- Ensure that you configured the mqweb server for use by the administrative REST API, the administrative REST API for MFT, the messaging REST API, or IBM MQ Console.For more information about configuring the mqweb server with a basic registry, see Basic configuration for the mqweb server.
- If the mqweb server is already configured, ensure that you added the appropriate users to enable messaging in step 5 of Basic configuration for the mqweb server. Users of the messaging REST API must be a member of the `MQWebUser` role. The `MQWebAdmin` and `MQWebAdminRO` roles are not applicable for the messaging REST API.
  - If **`mqRestMessagingAdoptWebUserContext`** is set to `true` in the mqweb server configuration, the users in the `MQWebUser` role must be authorized to access queues and topics that are used for messaging through OAM or RACF.
  - If **`mqRestMessagingAdoptWebUserContext`** is set to `false` in the mqweb server configuration, the user ID that is used to start the mqweb server must be authorized to access queues and topics that are used for messaging through OAM or RACF.
- Ensure that the messaging REST API is configured to connect to remote queue managers. For more information, see Configuring the connection mode for the messaging REST API.

About this task

You can connect to remote queue managers by using the messaging REST API. A remote queue manager can be a queue manager on another system, a queue manager in another installation, or a queue manager in the same installation as the mqweb server.

To connect to a remote queue manager, you must complete the following configuration steps:

- Configure a server-connection channel and a listener.
- Give authority to an appropriate user to access the queue manager.
- Create a CCDT file that contains the connection information for the queue manager.
- Add the connection information to the messaging REST API by using the **setmqweb remote** command.

Then, you can use the remote queue manager by providing the unique name in the resource URL in place of the queue manager name.

You can also configure your remote queue managers as part of a queue manager group. For more information, see "Setting up a queue manager group to use with the messaging REST API" on page 688.

Procedure

1. On the remote queue manager, create a server-connection channel to allow remote connections to the queue manager. You can create server-connection channels by using the **DEFINE CHANNEL** MQSC command on the command line.
   For example, to create a server-connection channel `QM1.SVRCONN` for queue manager QM1, enter the following command:

   ```
   DEFINE CHANNEL(QM1.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
   ```

   For more information about **DEFINE CHANNEL** and the options available, see DEFINE CHANNEL.
2. Ensure that an appropriate user is authorized to access the queue manager. This user must also be authorized to access any queues or topics that you use for messaging. The user needs `connect`, `inquire`, `alternate user`, and `set context` authority on the queue manager. On UNIX, Linux, and Windows use the **setmqaut** control command on the command line. On z/OS, define RACF profiles to give the authorized user access to the queue manager.
   For example, on UNIX, Linux, and Windows, to authorize a user, `exampleUser`, to access the queue manager QM1, enter the following command:

   ```
   setmqaut -m QM1 -t qmgr -p exampleUser +connect +inq +altusr +setall
   ```

   For more information about which user needs to be authorized, see "Determining the security principal used by the messaging REST API" on page 691.
3. ALW
   If no listener exists on the remote queue manager, create a listener to accept incoming network connections by using the **DEFINE LISTENER** MQSC command on the command line.
   For example, to create a listener `REMOTE.LISTENER` on port 1414 for remote queue manager QM1, enter the following command:

   ```
   runmqsc QM1
   DEFINE LISTENER(REMOTE.LISTENER) TRPTYPE(TCP) PORT(1414)
   end
   ```
4. Ensure that the listener is running by using the **START LISTENER** MQSC command on the command line:

   ALW For example, on AIX, Linux, and Windows to start the listener `REMOTE.LISTENER` for queue manager QM1, enter the following command:

   ```
   runmqsc QM1
   START LISTENER(REMOTE.LISTENER)
   end
   ```

z/OS For example, on z/OS, to start the listener, enter the following command:

```
runmqsc QM1
START LISTENER TRPTYPE(TCP) PORT(1414)
end
```

The channel initiator address space must be started before you can start a listener on z/OS.

5. On the system where the mqweb server that hosts the messaging REST API is running, create or update a JSON CCDT file that contains the queue manager connection information.

   The CCDT file must include the `name`, `clientConnection`, and `type` information. You can optionally include additional information such as `transmissionSecurity` information. For more information about all the CCDT channel attribute definitions, see Complete list of CCDT channel attribute definitions.

   The following example shows a basic JSON CCDT file for a remote queue manager connection. It sets the name of the channel to the same name as the example server-connection channel created in step "1" on page 686. The connection port is set to the same value as the port that is used by the listener. The connection host is set to the hostname of the system on which the remote queue manager, QM1, is running.

```
{
  "channel": [{
      "name": "QM1.SVRCONN",
      "clientConnection": {
        "connection": [{
            "host": "example.com",
            "port": 1414
          }],
        "queueManager": "QM1"
      },
      "type": "clientConnection"
    }]
}
```

6. From the installation that is running the mqweb server that hosts the messaging REST API, use the **setmqweb remote** command to add the remote queue manager information to the mqweb server configuration.

   As a minimum, you must specify the following parameters:

   - **-qmgrName**, where you specify the name of the queue manager.
   - **-ccdtURL**, where you specify the CCDT URL for the queue manager.
   - **-uniqueName**, where you specify a unique name for the queue manager. The unique name is used to differentiate remote queue managers that might have the same name, and therefore must not exist to identify another queue manager.

   You can specify several other options, such as the username and password to use for the remote queue manager connection, or details of the truststore and keystore. For a full list of parameters that can be specified with the **setmqweb remote** command, see setmqweb remote.

   For example, to add the remote queue manager QM1, with the example CCDT file, enter the following command:

```
setmqweb remote add -uniqueName "RemoteQM1" -qmgrName "QM1" -ccdtURL "c:\myccdts\ccdt.json"
```

Results

The remote queue manager can be used with the messaging REST API by using the unique name in the resource URL in place of the queue manager name.

Example

The following example sets up the remote queue manager connection for a queue manager QM1. The IBM MQ Console authorized to administer the queue manager based on the authorization that is given to the

user exampleUser. The credentials of this user are provided to the IBM MQ Console when the **setmqweb remote** is used to configure the queue manager connection.

1. On the system where the remote queue manager QM1 is, a server-connection channel and a listener are created. The listener is started, and authorization is given for user `exampleUser` to connect to the queue manager and access a queue that is used for messaging:

```
runmqsc QM1
#Define the server connection channel that will accept connections from the Console
DEFINE CHANNEL(QM1.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
# Define the listener to use for the connection from the Console
DEFINE LISTENER(REMOTE.LISTENER) TRPTYPE(TCP) PORT(1414)
# Start the listener
START LISTENER(REMOTE.LISTENER)
end

#Set mq authorization for exampleUser to access the queue manager and a queue for messaging
setmqaut -m QM1 -t qmgr -p exampleUser +connect +inq +setall +dsp
setmqaut -m QM1 -t queue -p exampleUser -n EXAMPLEQ +put +get +browse +inq
```

2. On the system where the mqweb server is running, a `QM1_ccdt.json` file is created with the following connection information:

```
{
  "channel": [{
      "name": "QM1.SVRCONN",
      "clientConnection": {
        "connection": [{
            "host": "example.com",
            "port": 1414
          }],
        "queueManager": "QM1"
      },
      "type": "clientConnection"
    }]
}
```

3. On the system where the mqweb server is running, the connection information for queue manager QM1 is added to the mqweb server. The credentials for `exampleUser` are included in the connection information:

```
setmqweb remote add -uniqueName "MACHINEAQM1" -qmgrName "QM1" -ccdtURL
"c:\myccdts\QM1_ccdt.json" -username "exampleUser" -password "password"
```

4. The messaging REST API can connect to the remote queue manager QM1 by using the unique name for the queue manager connection in place of the queue manager name in the resource URL:

```
curl -k https://localhost:9443/ibmmq/rest/v2/messaging/qmgr/MACHINEAQM1/queue/EXAMPLEQ/
message -X POST -u myuser:mypassword -H "ibm-mq-rest-csrf-token: value" -H "Content-Type:
text/plain;charset=utf-8" --data "Hello World!"
```

V 9.4.0 Setting up a queue manager group to use with the messaging REST API

You can use the messaging REST API to connect to queue manager groups for messaging. Before you can connect to a queue manager group, you must set up the remote queue manager configuration for the group. Then, you can connect to the queue manager group by using the unique name that is defined in the configuration information.

Before you begin

- Ensure that you configured the mqweb server for use by the administrative REST API, the administrative REST API for MFT, the messaging REST API, or IBM MQ Console.For more information about configuring the mqweb server with a basic registry, see Basic configuration for the mqweb server.
- If the mqweb server is already configured, ensure that you added the appropriate users to enable messaging in step 5 of Basic configuration for the mqweb server. Users of the messaging REST API must

be a member of the MQWebUser role. The MQWebAdmin and MQWebAdminRO roles are not applicable for the messaging REST API.

  - If **mqRestMessagingAdoptWebUserContext** is set to true in the mqweb server configuration, the users in the MQWebUser role must be authorized to access queues and topics that are used for messaging. You can authorize these users through OAM or RACF.
  - If **mqRestMessagingAdoptWebUserContext** is set to false in the mqweb server configuration, the user ID that starts the mqweb server must be authorized to access queues and topics that are used for messaging. You can authorize this user through OAM or RACF.

- Ensure that the messaging REST API is configured to connect to remote queue managers. For more information, see Configuring the connection mode for the messaging REST API

About this task

A queue manager group enables you to connect applications to any queue manager within the group. The group is defined as a set of connections in a client channel definition table (CCDT). When you use an MQCONN or an MQCONNX call, you reference the group by prefixing an asterisk to the queue manager name. With the messaging REST API, you reference the group by using the unique name that is associated with the queue manager group. The unique name is included in the resource URL in place of the queue manager name. For more information about queue manager groups, see "Queue manager groups in the CCDT" on page 890.

You can also configure your remote queue managers individually. For more information, see "Setting up a remote queue manager to use with the messaging REST API" on page 685.

Procedure

1. On each of the remote queue managers in the group, create a server-connection channel to allow remote connections to the queue manager. You can use the **DEFINE CHANNEL** MQSC command on the command line to create server-connection channels.
   For example, to create a server-connection channel QM1.SVRCONN for queue manager QM1, enter the following command:

   ```
   DEFINE CHANNEL(QM1.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
   ```

   For more information about **DEFINE CHANNEL** and the options available, see DEFINE CHANNEL.
2. On each of the remote queue managers in the group, ensure that an appropriate user is authorized to access the queue manager. This user must also be authorized to access any queues or topics that you use for messaging. The user needs connect, inquire, alternate user, and set context authority on the queue manager. On UNIX, Linux, and Windows use the **setmqaut** control command on the command line. On z/OS, define RACF profiles to give the authorized user access to the queue manager.
   For example, on UNIX, Linux, and Windows, enter the following command to authorize a user, exampleUser, to access the queue manager QM1:

   ```
   setmqaut -m QM1 -t qmgr -p exampleUser +connect +inq +altusr +setall
   ```

   For more information about which user needs to be authorized, see "Determining the security principal used by the messaging REST API" on page 691.
3. ALW
   If no listener exists on each of the remote queue managers in the group, create listeners to accept incoming network connections. You can use the **DEFINE LISTENER** MQSC command on the command line to create listeners.

For example, to create a listener REMOTE.LISTENER on port 1414 for remote queue manager QM1, enter the following command:

```
runmqsc QM1
DEFINE LISTENER(REMOTE.LISTENER) TRPTYPE(TCP) PORT(1414)
end
```

4. On each of the remote queue managers in the group, ensure that the listener is running by using the **START LISTENER** MQSC command on the command line.

   ALW For example, on AIX, Linux, and Windows to start the listener REMOTE.LISTENER for queue manager QM1, enter the following command:

   ```
   runmqsc QM1
   START LISTENER(REMOTE.LISTENER)
   end
   ```

   z/OS For example, on z/OS, to start the listener, enter the following command:

   ```
   runmqsc QM1
   START LISTENER TRPTYPE(TCP) PORT(1414)
   end
   ```

   The channel initiator address space must be started before you can start a listener on z/OS.

5. On the system where the mqweb server that hosts the messaging REST API is running, create a JSON CCDT file. This JSON file contains connection information for each queue manager in the group.

   The CCDT file must include the name, `clientConnection`, and `type` information for each queue manager connection. You can optionally include additional information such as `transmissionSecurity` information. For more information about all the CCDT channel attribute definitions, see Complete list of CCDT channel attribute definitions.

   The following example shows a basic JSON CCDT file for two queue manager connections. The first connection is for queue manager QM1. It has a server connection channel of QM1.SVRCONN, a listener on port 1414, and runs on host QM1.example.com. The second connection is for queue manager QM2. It has a server connection channel of QM2.SVRCONN, a listener on port 1415, and runs on host QM2.example.com. However, as the connections are part of queue manager group QMGRP, the **queueManager** field for both connections is set to the name of the queue manager group.

   ```
   {
     "channel": [{
         "name": "QM1.SVRCONN",
         "clientConnection": {
           "connection": [{
               "host": "QM1.example.com",
               "port": 1414
             }],
           "queueManager": "QMGRP"
         },
         "type": "clientConnection"
     }],
     "channel": [{
         "name": "QM2.SVRCONN",
         "clientConnection": {
           "connection": [{
               "host": "QM2.example.com",
               "port": 1415
             }],
           "queueManager": "QMGRP"
         },
         "type": "clientConnection"
      }]
   }
   ```

6. From the installation that is running the mqweb server that hosts the messaging REST API, use the **setmqweb remote** command to add the queue manager group to the mqweb server configuration.

   As a minimum, you must specify the following parameters:

   - **-qmgrName**, where you specify the group name for the queue manager group.

- **-ccdtURL**, where you specify the CCDT URL for the queue managers.
- **-uniqueName**, where you specify a unique name to identify the queue manager group.
- **-group**, to set the queue manager information as being for a group.

You can specify several other options, such as the username and password to use for the connection, or details of the truststore and keystore. For a full list of parameters that can be specified with the **setmqweb remote** command, see setmqweb remote.

For example, to add the queue manager group QMGRP, with the example CCDT file, enter the following command:

```
setmqweb remote add  -uniqueName "MyQMGRP" -qmgrName "QMGRP" -ccdtURL
"c:\myccdts\group_ccdt.json" -group
```

Results

The remote queue manager group can be used with the messaging REST API by using the unique name in the resource URL. A queue manager from the group is selected to complete the request, and information about which queue manager completed the request is returned in the response header `ibm-mq-resolved-qmgr`.

V 9.4.0 Determining the security principal used by the messaging REST API

When you use the messaging REST API, an appropriate user must be authorized to access the queue managers, queues, and topics that you want to connect to for messaging. The user that needs to be authorized depends on how your mqweb server is configured, and whether you are using remote queue managers with the messaging REST API.

By default, the security principal that is used to authorize access to the queue manager is the user that starts the mqweb server that runs the messaging REST API. The security principal that is used to authorize access to the queues and topics is the user that is logged in to the messaging REST API. However, your mqweb server or remote queue manager connection might be configured such that a different security principal is used.

Determining the security principal that is used to connect to the queue manager

For local queue manager connections, the security principal that is used to connect to the queue manager is the user that starts the mqweb server that runs the messaging REST API. For remote queue manager connections, the following security principals are used by the messaging REST API to authorize access to the queue manager, in order of priority. That is, if users are specified in multiple ways within the remote queue manager configuration, the first in the list is used for authorization.

1. The security principal is an adopted user context from a security exit.
2. The security principal is an adopted user context in a CHLAUTH rule on the server-connection channel that is used to connect to the remote queue manager.
3. The security principal is the user ID that is included in the remote queue manager configuration for the messaging REST API. This user ID is optionally included in the queue manager connection information when you add the queue manager with the **setmqweb remote** command.
4. The security principal is the user that starts the mqweb server that runs the messaging REST API.

For more information about setting up remote queue managers to use with the messaging REST API, see “Setting up a remote queue manager to use with the messaging REST API” on page 685.

Determining the security principal that is used to connect to queues and topics

You can set a property in the mqweb server configuration to determine what security principal is used to authorize connections to queues and topics when you use the messaging REST API. This property is the **mqRestMessagingAdoptWebUserContext** property. You can view what this property is set to by using the **dspmqweb properties** command.

- If **mqRestMessagingAdoptWebUserContext** is set to true, then the messaging REST API uses the user ID of the user that is logged in to the messaging REST API for authorization. Therefore, the user ID or user IDs that exist in the mqweb server configuration for use with the messaging REST API are the security principals that must be authorized to access the queues and topics.
- If **mqRestMessagingAdoptWebUserContext** is set to false, then the messaging REST API uses the user ID of the user that started the mqweb server that hosts the messaging REST API for authorization. Therefore, a user ID that is the same as the user ID that starts the mqweb server that hosts the messaging REST API must be authorized to access the queues and topics.

  If your queues and topics are on a remote queue manager, the security principal that is used for authorization might be determined by settings in the queue manager configuration. The following security principals might used, in order of priority:

  1. The security principal is an adopted user context from a security exit.
  2. The security principal is an adopted user context in a CHLAUTH rule on the server-connection channel that is used to connect to the remote queue manager. For example, you can configure a CHLAUTH rule on the server-connection channel to use the MCAUSER parameter. Then, all connections are mapped to a user ID that is authorized to use the queue manager.
  3. The security principal is an adopted user context from the AUTHINFO of the queue manager. If AUTHINFO object that is referred to by the CONNAUTH attribute of the queue manager is configured to use **ADOPTCTX(yes)**, then the security principal that is used to authorize connections to the queue manager is also used to authorize the queues and topics. For example, this security principal might be the user ID that is included in the remote queue manager connection information as part of the **setmqweb remote** command.

Related information

CHLAUTH

CONNAUTH

dspmqweb properties

Developing MQI applications with IBM MQ

IBM MQ provides support for C, Visual Basic, COBOL, Assembler, RPG, pTAL, and PL/I. These procedural languages use the message queue interface (MQI) to access message queuing services.

For detailed information about how to write your applications in your chosen language, see the subtopics.

For an overview of the call interface for procedural languages, see Call descriptions. This topic contains a list of the MQI calls, and each call shows you how to code the calls in each of these languages.

IBM MQ provides data definition files to help you to write your applications. For a full description, see “IBM MQ data definition files” on page 693.

To help you choose which procedural 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 languages. 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.

Related concepts

“Object-oriented applications” on page 15

IBM MQ provides support for JMS, Java, C++, and .NET. These languages and frameworks use the IBM MQ Object Model, which provides classes that provide the same functionality as IBM MQ calls and structures.

Technical overview

"Application development concepts" on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

Related reference

Developing applications reference

IBM MQ data definition files

IBM 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 IBM MQ COPY, header, include, and module files.

The data definition files to help you to write installable services exits are described in "User exits, API exits, and IBM MQ installable services" on page 902.

For data definition files supported on C++, see Using C++.

IBM i

For data definition files supported on RPG, see the IBM i 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

z/OS The name **thlqual** is the high-level qualifier of the installation library on z/OS.

This topic introduces IBM MQ data definition files, under these headings:

- "C language include files" on page 694
- "Visual Basic module files" on page 694

- “COBOL copy files” on page 694
- z/OS “System/390 assembler-language macros” on page 695
- z/OS “PL/I include files” on page 696

C language include files

The IBM MQ C include files are listed in C header files. They are installed in the following directories or libraries:

Platform	Installation directory or library
IBM i IBM i	QMQM/H
Linux AIX AIX and Linux	*MQ_INSTALLATION_PATH*/inc/
Windows Windows	*MQ_INSTALLATION_PATH*\Tools\c\include
z/OS z/OS	**thlqual**.SCSQC370

where *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

Note: For AIX and Linux, the include files are symbolically linked into `/usr/include`.

For more information about the structure of directories, see Planning file system support.

Visual Basic module files

IBM MQ for Windows provides four Visual Basic module files.

They are listed in Visual Basic module files and installed in

```
MQ_INSTALLATION_PATH\Tools\Samples\VB\Include
```

COBOL copy files

For COBOL, IBM 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).

IBM i Copy files containing data and interface definitions for IBM i 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 IBM MQ COBOL copy files are listed in COBOL COPY files. They are installed in the following directories:

Platform	Installation directory or library
AIX and Linux	*MQ_INSTALLATION_PATH*/inc/
IBM i	QMQM/QCBLLESRC
Windows	*MQ_INSTALLATION_PATH*\Tools\cobol\copybook (for Micro Focus COBOL) *MQ_INSTALLATION_PATH*\Tools\cobol\copybook\VAcobol (for IBM VisualAge® COBOL)
z/OS	**thlqual**.SCSQCOBC

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

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 Data types used in the MQI. 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

z/OS

IBM MQ for z/OS provides two assembler-language macros containing the named constants, and one macro to generate each structure.

They are listed in z/OS Assembler COPY files 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

z/OS

IBM MQ for z/OS provides include files that contain all the definitions that you need when you write IBM MQ applications in PL/I.

The files are listed in PL/I include files and installed in the **thlqual**.SCSQPLIC directory:

Include these files in your program if you are going to link the IBM MQ stub to your program (see “Preparing your program to run” on page 984). Include only CMQP if you intend to link the IBM MQ calls dynamically (see “Dynamically calling the IBM MQ stub” on page 990). Dynamic linking can be performed for batch and IMS programs only.

Writing a procedural application for queuing

Use this information to learn about writing queuing applications, connecting to and disconnecting from a queue manager, publish/subscribe, and opening and closing objects.

Use the following links to find out more about writing applications:

- “The Message Queue Interface overview” on page 697
- “Connecting to and disconnecting from a queue manager” on page 709
- “Opening and closing objects” on page 716
- “Putting messages on a queue” on page 727
- “Getting messages from a queue” on page 741
- “Writing publish/subscribe applications” on page 780
- “Inquiring about and setting object attributes” on page 820
- “Committing and backing out units of work” on page 823
- “Starting IBM MQ applications using triggers” on page 834
- “Working with the MQI and clusters” on page 852
- z/OS “Using and writing applications on IBM MQ for z/OS” on page 857
- z/OS “IMS and IMS bridge applications on IBM MQ for z/OS” on page 66

Related concepts

“Application development concepts” on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

“Developing applications for IBM MQ” on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

“Design considerations for IBM MQ applications” on page 47
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 IBM MQ.

“Writing client procedural applications” on page 879
What you need to know to write client applications on IBM MQ using a procedural language.

“Building a procedural application” on page 962

You can write an IBM MQ application in one of several procedural languages, and run the application on several different platforms.

"Handling procedural program errors" on page 999
This information explains errors associated with your applications MQI calls either when it makes a call, or when its message is delivered to its final destination.

Related tasks

"Using the IBM MQ sample procedural programs" on page 1018
These sample programs are written in procedural languages, and demonstrate typical uses of the Message Queue Interface (MQI). IBM MQ programs on different platforms.

The Message Queue Interface overview

Learn about the Message Queue Interface (MQI) components.

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

z/OS IBM 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 IBM 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 "Using the sample programs for z/OS" on page 1118.

IBM i IBM MQ for IBM i also supplies:

- *Data definition files* (sometimes known as copy files, macros, include files, and header files) that define the values of constants supplied with IBM MQ for IBM i.
- 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 IBM i platform.

AIX, Linux, and Windows systems also supply:

- Calls through which IBM MQ for AIX, Linux, and Windows 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. For further information about these samples, see "Using the sample programs on Multiplatforms" on page 1018.
- Sample source and executable code for bindings to external transaction managers.

Use the following links to find out more about the MQI:

- "MQI calls" on page 698
- "Sync point calls" on page 699
- "Data conversion, data types, data definitions, and structures" on page 700
- "IBM MQ stub programs and library files" on page 700
- "Parameters common to all the calls" on page 705
- "Specifying buffers" on page 706
- z/OS "z/OS batch considerations" on page 706

- "AIX and Linux signal handling" on page 707

Related concepts

"Connecting to and disconnecting from a queue manager" on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Opening and closing objects" on page 716
This information provides an insight into opening and closing IBM MQ objects.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Getting messages from a queue" on page 741
Use this information to learn about getting messages from a queue.

"Inquiring about and setting object attributes" on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

"Committing and backing out units of work" on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Working with the MQI and clusters" on page 852
There are special options on calls and return codes that relate to clustering.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

"IMS and IMS bridge applications on IBM MQ for z/OS" on page 66
This information helps you to write IMS applications using IBM MQ.

MQI calls

Use this information to learn about calls in the Message Queue Interface (MQI).

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.

z/OS 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 IBM 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. IBM i IBM i commitment controller is used to coordinate global units of work on IBM MQ for IBM i. 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.

See Call descriptions for a description of the MQI calls.

Sync point calls

Use this information to find out about sync point calls on different platforms.

Sync point calls are available as follows:

IBM MQ for z/OS calls

z/OS

IBM 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 sync point 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 sync point 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 827.)

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

IBM i calls

IBM i

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

IBM MQ calls on AIX, Linux, and Windows platforms

ALW

IBM MQ for AIX, Linux, and Windows provide the MQCMIT and MQBACK calls.

Use sync point calls in programs to tell the queue manager that all the MQGET and MQPUT operations since the last sync point 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, data types, data definitions, and structures

Use this information to learn about data conversions, elementary data types, IBM MQ data definitions, and structures when using the Message Queue Interface.

Data conversion

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

See MQXCNVC - Convert characters for the syntax used with the MQXCNVC call, and "Writing data-conversion exits" on page 947 for guidance on writing and invoking data conversion exits.

Elementary data types

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

These data types are described fully in Elementary data types.

IBM MQ data definitions

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

IBM i IBM MQ for IBM i 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 IBM MQ contain:

- Definitions of all the IBM MQ constants and return codes
- Definitions of the IBM 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 IBM MQ data definition files, see "IBM MQ data definition files" on page 693.

Structures

Structures, used with the MQI calls listed in "MQI calls" on page 698, are supplied in data definition files for each of the supported programming languages.

See Structure data types for a summary of the structures.

z/OS IBM i IBM MQ for z/OS and IBM MQ for IBM i supply files that contain constants for you to use when completing some of the fields of these structures. For more information about these, see IBM MQ data definitions.

IBM 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 "Building a procedural application" on page 962. For information about linking to C++ library files, see Using C++ *IBM MQ Using C++*.

AIX *IBM MQ for AIX library files*

On IBM 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, link to one of the following libraries:

Table 106. Library files for non-threaded AIX applications

Library file	Environment
libmqm.a	Server for C
libmqic.a and libmqm.a	Client for C
libmqmzf.a	Installable service exits for C
libmqmxa.a	Server XA interface
libmqmxa64.a	Server alternative XA interface
libmqcxa.a	Client XA interface
libmqcxa64.a	Client alternative XA interface
libmqmcbrt.o	IBM MQ runtime library for Micro Focus COBOL support
libmqmcb.a	Server for COBOL
libmqicb.a	Client for COBOL
libimqc23ia.a	Client for C++ (XLC 16)
libimqs23ia.a	Server for C++ (XLC 16)
V 9.4.0 libimqc23ca.a	Client for C++ (XLC 17)
V 9.4.0 libimqs23ca.a	Server for C++ (XLC 17)

V 9.4.0 Libraries containing "ia" have been built with the XLC 16 compiler, whilst libraries with "ca" in the name have been built with the XLC 17 compiler.

In a threaded application, link to one of the following libraries:

Table 107. Library files for threaded AIX applications.

A two column table listing the library files and the environment for each library file.

Library file	Environment
libmqm_r.a	Server for C
libmqic_r.a and libmqm_r.a	Client for C
libmqmzf_r.a	Installable service exits for C
libmqmxa_r.a	Server XA interface
libmqmxa64_r.a	Server alternative XA interface
libmqcxa_r.a	Client XA interface
libmqcxa64_r.a	Client alternative XA interface
libimqc23ia_r.a	Client for C++ (XLC 16)
libimqs23ia_r.a	Server for C++ (XLC 16)

Table 107. Library files for threaded AIX applications. A two column table listing the library files and the environment for each library file. *(continued)*	
Library file	**Environment**
V 9.4.0 libimqc23ca_r.a	Client for C++ (XLC 17)
V 9.4.0 libimqs23ca_r.a	Server for C++ (XLC 17)

V 9.4.0 Libraries with names that include `ia` have been built with the XLC 16 compiler, whilst libraries with names that include `ca` have been built with the XLC 17 compiler.

Note: You cannot link to more than one library. That is, you cannot link to both a threaded and a non-threaded library at the same time.

IBM i *IBM MQ for IBM i library files*

In IBM MQ for IBM i, 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:

Table 108. Library files for non-threaded IBM i applications	
Library file	**Environment**
LIBMQM	Server and Client 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:

Table 109. Library files for threaded IBM i applications	
Library file	**Environment**
LIBMQM_R	Server & client service program
IMQB23I4_R	C++ base service program
IMQS23I4_R	C++ server service program
LIBMQMZF_R	Installable exits for C
LIBMQIC_R	Client service program

On IBM MQ for IBM i, 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 Using C++.

Linux *IBM MQ for Linux library files*

On IBM 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, link to one of the following libraries:

Table 110. Library files for non-threaded Linux applications	
Library file	**Environment**
libmqm.so	Server for C
libmqic.so and libmqm.so	Client for C
libmqmzf.so	Installable service exits for C
libmqmxa.so	Server XA interface
libmqmxa64.so	Server alternative XA interface
libmqcxa.so	Client XA interface
libmqcxa64.so	Client alternative XA interface
libimqc23gl.so	Client for C++
libimqs23gl.so	Server for C++

In a threaded application, link to one of the following libraries:

Table 111. Library files for threaded Linux applications	
Library file	**Environment**
libmqm_r.so	Server for C
libmqic_r.so and libmqm_r.so	Client for C
libmqmzf_r.so	Installable service exits for C
libmqmxa_r.so	Server XA interface
libmqmxa64_r.so	Server alternative XA interface
libmqcxa_r.so	Client XA interface
libmqcxa64_r.so	Client alternative XA interface
libimqc23gl_r.so	Client for C++
libimqs23gl_r.so	Server for C++

Note: You cannot link to more than one library. That is, you cannot link to both a threaded and a non-threaded library at the same time.

Windows *IBM MQ for Windows library files*

On IBM 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:

Table 112. Library files for Windows applications	
Library File	**Environment**
MQ_INSTALLATION_PATH\Tools\Lib\mqm.lib	Server for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqic.lib	Client for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqmxa.lib	Server XA interface for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqcxa.lib	Client XA interface for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqicxa.lib	Client MTS for C (32-bit)

Table 112. Library files for Windows applications (continued)	
Library File	**Environment**
MQ_INSTALLATION_PATH\Tools\Lib\mqmcics4.lib32	Server TXSeries CICS support for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqccics4.lib32	Client TXSeries CICS support for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqmzf.lib	Installable services exits for C (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqmcbb.lib	Server for IBM COBOL (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqmcb.lib	Server for Micro Focus COBOL (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqiccbb.lib	Client for IBM COBOL (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\mqiccb.lib	Client for Micro Focus COBOL (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\imqs23vn.lib	Server for C++ (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\imqc23vn.lib	Client for C++ (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\imqb23vn.lib	Base for C++ (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib\imqx23vn.lib	Client MTS for C++ (32-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqm.lib	Server for C (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqic.lib	Client for C (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqmxa.lib	Server XA interface for C (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqcxa.lib	Client XA interface for C (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqicxa.lib	Client MTS for C (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqmcbb.lib	Server for IBM COBOL (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqmcb.lib	Server for Micro Focus COBOL (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqiccbb.lib	Client for IBM COBOL (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\mqiccb.lib	Client for Micro Focus COBOL (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\imqs23vn.lib	Server for C++ (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\imqc23vn.lib	Client for C++ (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\imqb23vn.lib	Base for C++ (64-bit)
MQ_INSTALLATION_PATH\Tools\Lib64\imqx23vn.lib	Client MTS for C++ (64-bit)

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Use `amqmdnet.dll` for compiling .NET programs. See "Compiling IBM MQ .NET programs" on page 591 within the section "Developing .NET applications" on page 535 for more information.

These files are shipped for compatibility with previous releases:

```
mqic32.lib
mqic32xa.lib
```

z/OS *IBM MQ for z/OS stub programs*

Before you can run a program written with IBM MQ for z/OS, you must link-edit it to the stub program supplied with IBM 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 IBM MQ for z/OS can process.

IBM 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 by way of 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

Attention: If you use a stub program other than one listed for a specific environment, it might have unpredictable results.

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 827.

Alternatively, you can dynamically call the stub from within your program. This technique is described in "Dynamically calling the IBM MQ stub" on page 990.

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

Do not run applications that are link-edited with CSQBSTUB and CSQQSTUB in the same IMS MPP region. This can cause problems such as DFS3607I or CSQQ005E messages. The first MQCONN call in an address space determines which interface is used, therefore CSQQSTUB and CSQBSTUB transactions must run in different IMS message regions.

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 a *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 an IBM 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 a call is successful, each call returns a *completion code* when the call is complete. The completion code is typically either MQCC_OK indicating success, or MQCC_FAILED indicating failure. 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. See Call descriptions and select the appropriate call from the list.

For more detailed information, including ideas for corrective action, see:

- z/OS IBM MQ for z/OS messages, completion, and reason codes for IBM MQ for z/OS
- Messages and reason codes for all other IBM MQ platforms

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.

z/OS *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.

Linux AIX *AIX and Linux considerations*

Considerations that you need to be aware of when developing AIX and Linux applications.

Linux AIX *The fork system call in AIX and Linux systems*

Note these considerations when using a fork system call in IBM MQ applications.

If your application wants to use `fork`, the parent process of that application should call `fork` before making any IBM MQ calls, for example, `MQCONN`, or creating an IBM MQ object using **`ImqQueueManager`**.

If your application wants to create a child process after making any IBM MQ calls, the application code must use a `fork()` with `exec()` to ensure that the child is a new instance, and not an exact copy of the parent.

If your application does not use `exec()`, the IBM MQ API call made within the child process returns MQRC_ENVIRONMENT_ERROR.

Linux AIX *AIX and Linux signal handling*

In general, AIX and Linux systems have moved from a nonthreaded (process) environment to a multithreaded environment. In many instances, signals and signal handling, although supported, do not fit well into the multithreaded environment and various restrictions exist.

In general, AIX and Linux 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 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.

Linux AIX *Unthreaded applications*

Each MQI function sets up its own signal handler for the signals. 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.

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 IBM 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 IBM MQ from MQCONN (or MQCONNX) until MQDISC.

Synchronous signals

Synchronous signals arise in a specific thread.

AIX and Linux systems safely allow the setting up of a signal handler for such signals for the whole process. However, IBM MQ sets up its own handler for the following signals, in the application process, while any thread is connected to IBM MQ:

SIGBUS
SIGFPE
SIGSEGV
SIGILL

If you are writing multithreaded applications, there is only one process-wide signal handler for each signal. When IBM MQ sets up its own synchronous signal handlers it saves any previously registered handlers for each signal. After IBM MQ handles one of the signals listed, IBM MQ attempts to call the signal handler that was in effect at the time of the first IBM MQ connection within the process. The previously registered handlers are restored when all application threads have disconnected from IBM MQ.

Because signal handlers are saved and restored by IBM MQ, application threads must not establish signal handlers for these signals while there is any possibility that another thread of the same process is also connected to IBM 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 IBM MQ, the application's signal handler must call the corresponding IBM 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 IBM MQ, the previous signal handler must be restored before the application disconnects from IBM MQ.
- When an application establishes a signal handler before connecting to IBM MQ, the application must disconnect from IBM 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

IBM MQ does not use any asynchronous signals in threaded applications unless they are client applications.

Additional considerations for threaded client applications

IBM 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)

Linux AIX *Additional considerations when using AIX and Linux signal handling in MQI*

When using MQI for signal handling on AIX and Linux, there are additional considerations for fastpath applications, MQI function calls within signal handlers, signals during MQI calls, user exits and installable services, and VMS exit handlers.

Fastpath (trusted) applications

Fastpath applications run in the same process as IBM MQ and so are running in the multithreaded environment.

In this environment IBM 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 IBM 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 712.

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.

For C++ destructor methods, which might 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(), IBM MQ backs out uncommitted messages in sync point 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.

AIX In the AIX environment, IBM 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 IBM 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 an IBM MQ process in a multithreaded environment have the same restrictions as for fastpath applications. Consider these to be permanently connected to IBM MQ and so not using signals or non-threadsafe operating system calls.

Connecting to and disconnecting from a queue manager

To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

The way that this connection is made depends on the platform and the environment in which the program is operating:

Multi **IBM MQ for Multiplatforms**

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.

z/OS **IBM MQ for z/OS batch**

Programs that run in this environment 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.

z/OS batch programs can connect, consecutively or concurrently, to multiple queue managers on the same TCB.

z/OS **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 about how to control the IMS adapter on z/OS, see Administering IBM MQ for z/OS. 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.

Following an IMS call that establishes a syncpoint, and before processing a message for another user, the IMS adapter ensures that the application closes handles and disconnects from the queue manager. See “Syncpoints in IMS applications” on page 826.

IMS programs can connect, consecutively or concurrently, to multiple queue managers on the same TCB.

z/OS **CICS Transaction Server for z/OS**

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 typically made automatically at initialization, but you can also use the CKQC transaction that is supplied with IBM MQ for z/OS. For more information about CKQC, see Administering IBM MQ for z/OS.

CICS tasks can connect only to the queue manager that the CICS region is connected to.

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.

Use the following links to find out more about connecting and disconnecting from a queue manager:

- “Connecting to a queue manager using the MQCONN call” on page 711
- “Connecting to a queue manager using the MQCONNX call” on page 712
- “Disconnecting programs from a queue manager using MQDISC” on page 716

Related concepts

“The Message Queue Interface overview” on page 697
Learn about the Message Queue Interface (MQI) components.

“Opening and closing objects” on page 716
This information provides an insight into opening and closing IBM MQ objects.

“Putting messages on a queue” on page 727
Use this information to learn how to put messages on a queue.

“Getting messages from a queue” on page 741
Use this information to learn about getting messages from a queue.

“Inquiring about and setting object attributes” on page 820

Attributes are the properties that define the characteristics of an IBM MQ object.

“Committing and backing out units of work” on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

“Starting IBM MQ applications using triggers” on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

“Working with the MQI and clusters” on page 852
There are special options on calls and return codes that relate to clustering.

“Using and writing applications on IBM MQ for z/OS” on page 857
IBM 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.

“IMS and IMS bridge applications on IBM MQ for z/OS” on page 66
This information helps you to write IMS applications using IBM MQ.

Connecting to a queue manager using the MQCONN call

Use this information to learn how to connect 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:

- z/OS For IBM MQ for z/OS, in the batch environment, the default queue manager is specified in the CSQBDEFV module.
- Multi For IBM MQ for Multiplatforms, the default queue manager is specified in the mqs.ini file.

z/OS 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.

When you connect to a queue manager it must be local to the task. It must belong to the same system as the IBM MQ application.

z/OS 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 IBM 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.

An application must be authorized for it to successfully connect to a queue manager. For more information, see Securing.

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 must not issue the MQDISC call in this situation because the calling application expects to remain connected.

The scope of the connection handle is the same as the scope of the object handle (see “Opening objects using the MQOPEN call” on page 718).

Descriptions of the parameters are given in the description of the MQCONN call in MQCONN.

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

The scope of an MQCONN or MQCONNX call is typically 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 IBM MQ calls, each one must issue MQCONN or MQCONNX.

It is not necessary for each call to be made to the same queue manager when a process makes multiple MQCONN calls. However, only one IBM MQ connection can be made from a thread at a time. Alternatively, consider "Shared (thread independent) connections with MQCONNX" on page 713 to allow multiple IBM MQ connections from a single thread and an IBM MQ connection to be used from any thread. [7]

If your application is running as a client, it can connect to more than one queue manager within a thread.

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 z/OS, or a queue sharing group name on z/OS shared queue systems. The options to control how the connection is made to the queue manager are supplied in a structure called the MQCNO.

The output from MQCONNX is:

- A connection handle (Hconn)
- A completion code
- A reason code

You use the connection handle on subsequent MQI calls.

The connect options, set in the *Options* field of the MQCNO structure, allows several attributes of the connection to be controlled. Of particular note are the following groups of options:

- The binding options allow *trusted applications* to be created. Trusted applications imply that the IBM 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 behavior is requested by specifying the MQCNO_FASTPATH_BINDING option. For more information about the restrictions that apply to trusted applications, see "Restrictions for trusted applications" on page 713.
- The handle-sharing options allow shared connections to be created. Shared connections can share handles between different threads within the same process. For more information about shared connections, see "Shared (thread independent) connections with MQCONNX" on page 713.

The MQCNO also allows the application to control how the connection to the queue manager is authenticated. Authentication credentials can be specified in an MQCSP structure that is referenced from the MQCNO structure.

For a full description of the parameters to the MQCONNX call, and the connection attributes that can be controlled, see MQCONNX - Connect queue manager (extended).

[7] When using multithreaded applications with IBM MQ for AIX or Linux systems you need to ensure that the applications have a sufficient stack size for the threads. Consider using a stack size of 256 KB, or larger, when multithreaded applications are making MQI calls, either by themselves or, with other signal handlers (for example, CICS).

Restrictions for trusted applications
Restrictions that apply to trusted applications. Some restrictions apply to all platforms and others are platform specific.

T

- 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.
- Linux AIX On AIX and Linux 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.
- IBM i On IBM i:
  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 IBM i profile swapping API functions, QSYGETPH, QWTSETP, and QSYRLSPH to temporarily change the current user of the job to QMQM while the IBM MQ programs run. Details of these functions, together with an example of their use, is provided in the Security APIs section of the IBM i*Application programming interfaces* documentation.
  2. Do not cancel trusted applications using System-Request Option 2, or by ending the jobs in which they are running using ENDJOB.
- ALW On AIX, Linux, and Windows 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.

Multi *Shared (thread independent) connections with MQCONNX*
Use this information to learn about Shared connections with MQCONNX, and some usage notes to consider.

Note: z/OS Not supported on IBM MQ for z/OS.

On Multiplatforms, 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. If your application is running in a transactional environment that requires MQI calls to be issued on the same thread, you must use the following default option:

MQCNO_HANDLE_SHARE_NONE
Creates a non-shared connection.

In most other environments, you can use one of the following thread independent, shared connection options:

MQCNO_HANDLE_SHARE_BLOCK
Creates a shared connection. On a `MQCNO_HANDLE_SHARE_BLOCK` connection, if the connection is currently in use by an MQI call on another thread, the MQI call waits until the current MQI call has completed.

MQCNO_HANDLE_SHARE_NO_BLOCK
Creates a shared connection. On a `MQCNO_HANDLE_SHARE_NO_BLOCK` connection, if the connection is currently in use by an MQI call on another thread, the MQI call fails immediately with a reason of `MQRC_CALL_IN_PROGRESS`.

Except for the MTS (Microsoft Transaction Server) environment, the default value is MQCNO_HANDLE_SHARE_NONE. In the MTS environment, the default value is MQCNO_HANDLE_SHARE_BLOCK.

A connection handle is returned from the MQCONNX call. The handle can be used by subsequent MQI calls from any thread in the process, associating those calls with the handle returned from the MQCONNX. MQI calls using a single shared handle are serialized across threads.

For example, the following sequence of activity is possible with a shared handle:

1. Thread 1 issues MQCONNX and gets a shared handle *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 handle 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 blocking can cause difficulties. Suppose that in step "2" on page 714, thread 1 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 on thread 1 takes. If you prefer that an MQI call returns with an error if another MQI call is already running on the handle, 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.

Multi *Use of MQCONNX call options with MQ_CONNECT_TYPE*

Use this information to understand the different MQCONNX call options and how they are used with the **MQ_CONNECT_TYPE** environment variable.

Note: MQ_CONNECT_TYPE only has any effect for STANDARD bindings. For other bindings, **MQ_CONNECT_TYPE** is ignored.

On IBM MQ for Multiplatforms, you can use the environment variable, **MQ_CONNECT_TYPE** in combination with the type of binding specified in the Options field of the MQCNO structure used on an MQCONNX call.

*Table 113. How MQCONNX call options are used with the **MQ_CONNECT_TYPE** environment variable*

MQCONNX call option	MQ_CONNECT_TYPE environment variable	Result
STANDARD	UNDEFINED	STANDARD
STANDARD	STANDARD	STANDARD
STANDARD	FASTPATH	STANDARD

*Table 113. How MQCONNX call options are used with the **MQ_CONNECT_TYPE** environment variable (continued)*

MQCONNX call option	MQ_CONNECT_TYPE environment variable	Result
STANDARD	CLIENT	CLIENT
STANDARD	LOCAL	STANDARD

If MQCNO_STANDARD_BINDING is not specified, you can use MQCNO_NONE, which defaults to MQCNO_STANDARD_BINDING.

Authentication and Identity for MQCONN and MQCONNX

Use this task to learn how applications can supply credentials that are used for authentication when they connect to IBM MQ.

The default user identity

When an application uses the message queue interface (MQI) to connect to IBM MQ with either MQCONN or MQCONNX, a user identity is always established and associated with the connection.

By default, the initial user identity is always that of the operating system process the application is running under. This initial identity might be sufficient for locally bound or trusted application connections.

When an application connects to a queue manager with an MQCONN call, the application cannot modify the default user ID. However, the following mechanisms can change the user ID that is associated with the connection:

- A client-side or server-side security exit.
- Channel authentication rules on the queue manager.
- A client user ID established during TLS mutual authentication.

Using MQCONNX to supply credentials

MQCONNX gives an application more control over the identity that is associated with the connection. An application can supply an MQCSP structure as part of the connect options that are specified in the **ConnectOpts** parameter to MQCONNX. The MQCSP structure can contain credentials that are used to establish a user identity. IBM MQ supports the following credentials in the MQCSP structure:

- A user ID and password.
- V 9.4.0 From IBM MQ 9.3.4, an authentication token, if the application connects to a queue manager that runs on AIX or Linux systems.

The queue manager's connection authentication and channel authentication configuration controls how the credentials that are supplied by an application are processed. For example, this configuration affects the following aspects:

- Whether the credentials in the MQCSP structure are validated, and how they are validated.
- Whether the user ID in the credentials in the MQCSP structure is mapped to another user ID.
- Whether the authenticated user is then adopted as the context for the application.

For more information about connection authentication, see Connection authentication. For more information about channel authentication, see Channel authentication records.

Several of the sample programs written in C that use the MQI demonstrate how the MQCSP structure is used to provide authentication credentials. For more information, see the following sample programs:

- “The Get sample programs” on page 1051
- “The Put sample programs” on page 1063

- “The Browser sample program” on page 1039
- “The TLS sample program” on page 1078

Related information

Identifying and authenticating users using the MQCSP structure
MQCSP - Security parameters
Identifying and authenticating users

Disconnecting programs from a queue manager using MQDISC

Use this information to learn about 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:

- z/OS 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.
- IBM i On IBM MQ for IBM i 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.

On CICS running on Multiplatforms, 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.

z/OS For a client connected to z/OS, when an MQDISC call is issued an implicit commit takes place, but any queue handles that are still open are not closed until the channel actually ends.

z/OS If you use MQCONNX to connect on IBM 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 MQDISC.

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 an IBM MQ object closes or disconnect from that object. Even if the authority of a job that has connected to or opened an IBM MQ object is revoked, the MQCLOSE and MQDISC calls are accepted.

Opening and closing objects

This information provides an insight into opening and closing IBM MQ objects.

To perform any of the following operations, you must first *open* the relevant IBM 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 735).

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 709.

There are four types of IBM 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 IBM MQ objects, see Object types.

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 736 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 IBM i 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.

Use the following links to find out more about opening and closing objects:

- "Opening objects using the MQOPEN call" on page 718
- "Creating dynamic queues" on page 725
- "Opening remote queues" on page 726
- "Closing objects using the MQCLOSE call" on page 726

Related concepts

"The Message Queue Interface overview" on page 697
Learn about the Message Queue Interface (MQI) components.

"Connecting to and disconnecting from a queue manager" on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Getting messages from a queue" on page 741
Use this information to learn about getting messages from a queue.

"Inquiring about and setting object attributes" on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

"Committing and backing out units of work" on page 823

This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Working with the MQI and clusters" on page 852
There are special options on calls and return codes that relate to clustering.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

"IMS and IMS bridge applications on IBM MQ for z/OS" on page 66
This information helps you to write IMS applications using IBM MQ.

Opening objects using the MQOPEN call

Use this information to learn about 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 Multiplatforms, 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 712 and "Shared (thread independent) connections with MQCONNX" on page 713. 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 z/OS batch environment, 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 MQOPEN.

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.

For information about using the MQOD structure for distribution lists, see "Using the MQOD structure" on page 738 under "Distribution lists" on page 736.

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 an IBM 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 114 on page 719 for an overview of the name resolution process. The name that you supply in *ObjectQMgrName* is resolved *before* that in *ObjectName*.

Table 114 on page 719 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 114. Resolving queue names when using MQOPEN

Input to MQOD	**Input to MQOD**	**Resolved names**	**Resolved names**	**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)
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)

Table 114. Resolving queue names when using MQOPEN (continued)

Input to MQOD	Input to MQOD	Resolved names	Resolved names	Resolved names
Blank or local queue manager	Alias queue with or without CLUSTER attribute	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 locally defined where the *ObjectQMgrName* is specified, but can resolve to a clustered alias (hosted on other queue managers) where the *ObjectQMgrName* is blank.		
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

Table 114. Resolving queue names when using MQOPEN (continued)				
Input to MQOD	**Input to MQOD**	**Resolved names**	**Resolved names**	**Resolved names**
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)
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)

Notes:

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. z/OS For IBM MQ for z/OS queue managers that are part of a queue sharing group (QSG), the name of the queue sharing group can be used instead of the local queue manager name in Table 114 on page 719. If the local queue manager cannot open the target queue, or put a message to the queue, the message is transferred to the specified ObjectQMgrName through, either intra-group queuing, or an IBM MQ channel.
6. In the *ObjectName* column of the table, CLUSTER refers to both the CLUSTER and CLUSNL attributes of the queue.
7. 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.
8. If you have assigned a different cluster transmission queue to each cluster-sender channel, SYSTEM.CLUSTER.TRANSMIT.QUEUE might not be the name of the cluster transmission queue.

For more information about multiple cluster transmission queues, see Clustering: Planning how to configure cluster transmission queues.

9. In the situation where the queue manager is not the name of any local object; cluster queue managers, or queue manager alias found.

   When you have supplied a queue manager name using **ObjectQMgrName**, and there are multiple cluster channels with different cluster names known by the local queue manager that would reach that destination, then any of these channels might be used to move the message, regardless of the cluster name of the destination queue.

   This might be unexpected, if you were anticipating messages for that queue only to be sent through a channel that has the same cluster name as the queue.

   However, the **ObjectQMgrName** takes precedence in this case, and cluster workload balancing takes into consideration all channels that might reach that queue manager, regardless of the cluster name they are in.

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.

While an alias queue is unable to resolve to another locally defined alias queue (shared in a cluster or not), resolving to a remotely defined cluster alias queue is permitted and can therefore be specified as the base queue.

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 What is queue name resolution?.

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 alternative 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
The binding used for the queue handle is taken from the **DefBind** queue attribute, which can take the value MQBND_BIND_ON_OPEN, MQBND_BIND_NOT_FIXED, or MQBND_BIND_ON_GROUP.

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.

To specify that all messages in a message groups put to a queue using MQPUT are allocated to the same destination instance, use the MQOO_BIND_ON_GROUP option on the MQOPEN call.

Either MQOO_BIND_ON_OPEN or MQOO_BIND_ON_GROUP must be specified when using message groups with clusters to ensure that all messages in the group are processed at the same destination.

If you do not specify any of these options the default, MQOO_BIND_AS_Q_DEF, is used.

If you specify the name of a queue manager in the MQOD, the queue at that queue manager is selected. If the queue manager name is blank, any instance can be selected. See "MQOPEN and clusters" on page 853 for more information.

If you open a cluster queue using a QALIAS definition, some queue attributes are defined by the alias queue, and not the base queue. Cluster attributes are among the attributes of the base queue definition that are overridden by the alias queue. For example, in the following snippet, the cluster queue is opened with MQOO_BIND_NOT_FIXED and not MQOO_BIND_ON_OPEN. The cluster queue definition is advertised throughout the cluster, the alias queue definition is local to the queue manager.

```
DEFINE QLOCAL(CLQ1) CLUSTER(MYCLUSTER) DEFBIND(OPEN) REPLACE
DEFINE QALIAS(ACLQ1) TARGET(CLQ1) DEFBIND(NOTFIXED) REPLACE
```

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 774.

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 736.
3. Use the MQOO_CO_OP in conjunction with MQOO_BROWSE if you are using cooperative browsing; see Options

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 115 on page 723 to see how the setting of these attributes affects the type of access that you will be given when you use this option.

Table 115. 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

Table 115. How queue attributes and options of the MQOPEN call affect access to queues (continued)				
Queue attributes		**Type of access with MQOPEN options**		
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 115 on page 723 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 736.

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 820).

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.

Related concepts
"Message context" on page 45
Message context information allows the application that retrieves the message to find out about the originator of the message.

"Controlling message context information" on page 733
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. You can use the options field in the MQPMO structure to control context information.

MQOPEN option for alternative 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 they are working for, 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 alternative 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.

z/OS *MQOPEN option for queue manager quiescing*

If you use the MQOPEN call when the queue manager is in a quiescing state, the call might fail, depending on which environment you are using.

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 and Multiplatforms 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 manager 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 728).

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 IBM 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.

There is more information on dynamic queues in Dynamic and Model queues.

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 inTable 114 on page 719.

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.
- z/OS On IBM 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 sync point, so you can close queues before or after sync point.

As input to the MQCLOSE call, you must supply:

- A connection handle. Use the same connection handle used to open it. For CICS applications on z/OS, you can alternatively 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 MQCLOSE.

Putting messages on a queue

Use this information to learn how to put 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 afterward, 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.

Use the following links to find out more about putting messages on a queue:

- “Putting messages on a local queue using the MQPUT call” on page 728
- “Putting messages on a remote queue” on page 732
- “Setting properties of a message” on page 733
- “Controlling message context information” on page 733
- “Putting one message on a queue using the MQPUT1 call” on page 735
- “Distribution lists” on page 736
- “Some cases where the put calls fail” on page 741

Related concepts

“The Message Queue Interface overview” on page 697
Learn about the Message Queue Interface (MQI) components.

“Connecting to and disconnecting from a queue manager” on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

“Opening and closing objects” on page 716
This information provides an insight into opening and closing IBM MQ objects.

“Getting messages from a queue” on page 741
Use this information to learn about getting messages from a queue.

“Inquiring about and setting object attributes” on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

“Committing and backing out units of work” on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

“Starting IBM MQ applications using triggers” on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

“Working with the MQI and clusters” on page 852
There are special options on calls and return codes that relate to clustering.

“Using and writing applications on IBM MQ for z/OS” on page 857
IBM 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.

“IMS and IMS bridge applications on IBM MQ for z/OS” on page 66

This information helps you to write IMS applications using IBM MQ.

Putting messages on a local queue using the MQPUT call

Use this information to learn about 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 as follows:

- 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 MQPUT.

For more description on the information needed as input to the MQPUT call, see the following links:

- “Specifying handles” on page 728
- “Defining messages using the MQMD structure” on page 728
- “Specifying options using the MQPMO structure” on page 729
- “The data in your message” on page 731
- “Putting messages: Using message handles” on page 732

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 Configuring a queue manager cluster 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 "IBM MQ messages" on page 17, and there is a description of the structure itself in MQMD.

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 MQPMO.

The structure includes the following fields:

- *StrucId*
- *Version*
- *Options*
- *Context*
- *ResolvedQName*
- *ResolvedQMgrName*
- *RecsPresent*
- *PutMsgRecsFields*
- *ResponseRecOffset and ResponseRecPtr*
- *OriginalMsgHandle*
- *NewMsgHandle*
- *Action*
- *PubLevel*

The contents of these fields is as follows:

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 736). 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.

The way that the call operates with sync points is determined by the platform. The sync point control default is yes for z/OS and no for Multiplatforms.

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 45. For information about using the MQPMO structure to control the context information in a message, see "Controlling message context information" on page 733.

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 736). 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 MQPMR and MQRR.

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 740.

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 740 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 739 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 Action (MQLONG).

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 Action (MQLONG).

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 IBM 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 IBM 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.

IBM MQ adds header information to messages in the following circumstances:

- When you put a message on a remote queue, IBM 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 IBM 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, IBM 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, IBM MQ might add an MQMDE.

These structures are described in MQDH and MQMDE.

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 Attributes for the queue manager.

z/OS For information on how undelivered messages are handled in a distributed queuing environment, see Undelivered/unprocessed messages.

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 Action (MQLONG). 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 726. 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 IBM MQ distributed queuing techniques.

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.

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.

If you are putting your message to a queue on a queue manager at a level earlier than IBM WebSphere MQ 7.0, which cannot process message properties, you can set the PropertyControl parameter in the channel definition to specify how the properties are to be treated.

Controlling message context information

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. You can use the options field in the MQPMO structure to control context information.

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

To control context information, use the *Options* field in the MQPMO structure.

If you do not specify any options for context information, 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. When putting a message with no context, any authority checks made by IBM MQ are made using a blank user ID. A blank user ID cannot be assigned explicit authority to IBM MQ resources but is treated as a member of the special group 'nobody'. For more details on the special group nobody, see Installable services interface reference information.

You can do context setting using MQOPEN followed by MQPUT using the MQOO_ option and MQPMO_ option indicated in the following sections. You can also do context setting using just an MQPUT1, in which case you just need to select the MQPMO_ option indicated in the sections below.

The following sections of this topic 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.

In order to modify any of the message context options, you must have the appropriate authorizations to issue the call. For example, in order to use MQOO_SET_IDENTITY_CONTEXT or MQPMO_SET_IDENTITY_CONTEXT, you must have `+setid` permission.

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.

Related concepts

"Message context" on page 45
Message context information allows the application that retrieves the message to find out about the originator of the message.

Related reference

"MQOPEN options relating to message context" on page 724
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.

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 718). 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 729).

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 Configuring a queue manager cluster for more information.

There is a description of the MQPUT1 call in MQPUT1.

Multi *Distribution lists*

On IBM MQ for Multiplatforms, distribution lists allow you to put a message to multiple destinations in a single MQPUT or MQPUT1 call. A single MQOPEN call can open multiple queues and a single MQPUT call can then put a message to each of those queues. 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.

Attention: Distribution lists do not support the use of alias queues that point to topic objects. If an alias queue points to a topic object in a distribution list, IBM MQ returns MQRC_ALIAS_BASE_Q_TYPE_ERROR.

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 56 on page 737 shows how distribution lists work.

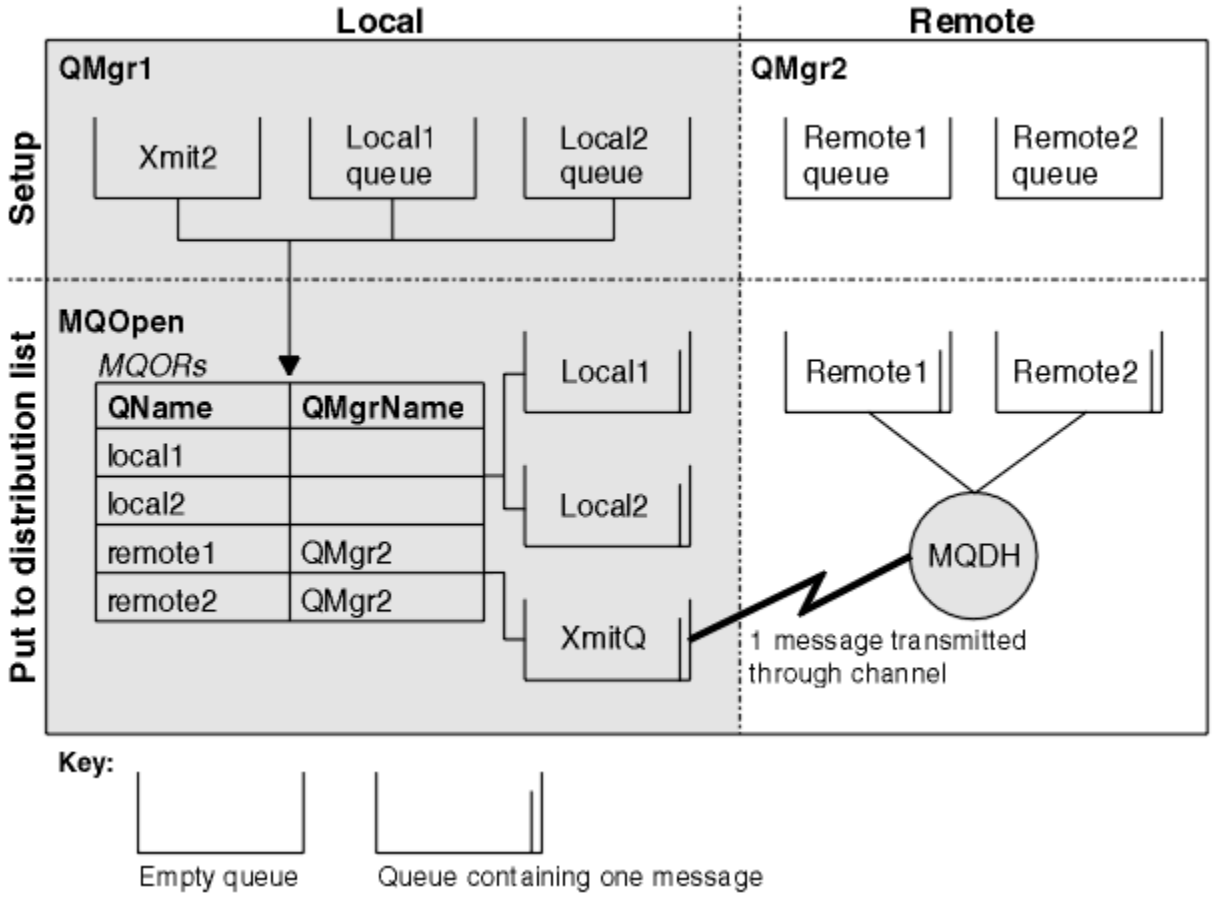


Figure 56. How distribution lists work

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 727 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 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 ObjectName and ObjectQMgrName 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 code 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 code 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 code 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 MQRR for further information about the MQRR structure.

Figure 57 on page 739 shows how you can open a distribution list in C.

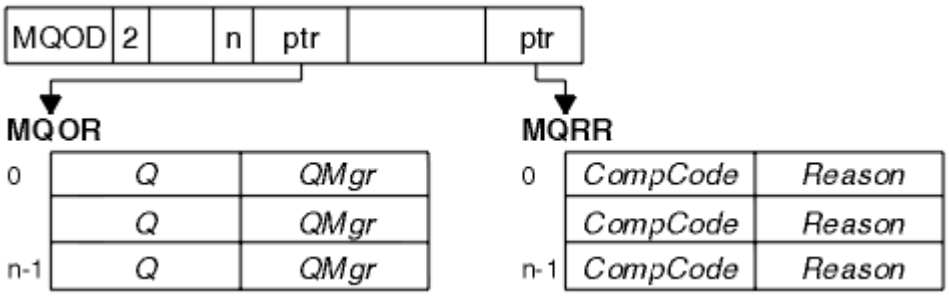


Figure 57. Opening a distribution list in C

Figure 58 on page 739 shows how you can open a distribution list in COBOL.

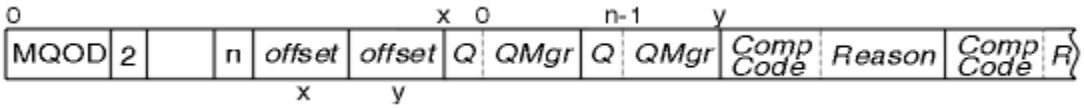


Figure 58. Opening a distribution list in COBOL

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 716 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 727 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 MQMD 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 729 for information about completing 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 MQPMR.

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 1049 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 59 on page 740 shows how you can put a message to a distribution list in C.

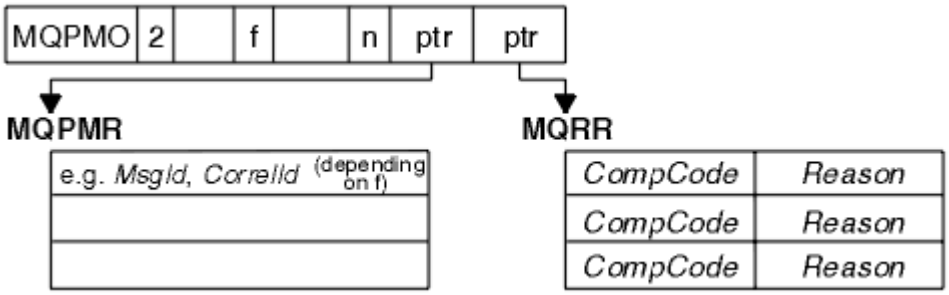


Figure 59. Putting a message to a distribution list in C

Figure 60 on page 741 shows how you can put a message to a distribution list in COBOL.

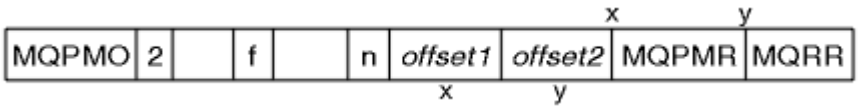


Figure 60. Putting a message to a distribution list in COBOL

Using MQPUT1

If you are using MQPUT1, consider the following points:

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 MQOPEN. 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

Use this information to learn about 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 709 and “Opening and closing objects” on page 716.

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.

Use the following links to find out more about getting messages from a queue:

- "Getting messages from a queue using the MQGET call" on page 742
- "The order in which messages are retrieved from a queue" on page 746
- "Getting a particular message" on page 758
- "Improving performance of non-persistent messages" on page 759
- z/OS "Type of index" on page 763
- "Handling messages greater than 4 MB long" on page 764
- "Waiting for messages" on page 769
- z/OS "Signaling" on page 770
- z/OS "Skipping backout" on page 771
- "Application data conversion" on page 773
- "Browsing messages on a queue" on page 774
- "Some cases where the MQGET call fails" on page 780

Related concepts

"The Message Queue Interface overview" on page 697
Learn about the Message Queue Interface (MQI) components.

"Connecting to and disconnecting from a queue manager" on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Opening and closing objects" on page 716
This information provides an insight into opening and closing IBM MQ objects.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Inquiring about and setting object attributes" on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

"Committing and backing out units of work" on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Working with the MQI and clusters" on page 852
There are special options on calls and return codes that relate to clustering.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

"IMS and IMS bridge applications on IBM MQ for z/OS" on page 66
This information helps you to write IMS applications using IBM MQ.

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 MQGET.

The following sections describe the information you must supply as input to the MQGET call.

- "Specifying connection handles" on page 743
- "Describing messages using the MQMD structure and the MQGET call" on page 743
- "Specifying MQGET options using the MQGMO structure" on page 744
- "Specifying the size of the buffer area" on page 746

Specifying connection handles

z/OS 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 "IBM MQ messages" on page 17, and there is a description of the structure itself in MQMD.

If you know which message you want to get from the queue, see "Getting a particular message" on page 758.

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 746 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 MQGMO.

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 769)
- Whether the get operation is included in a unit of work.
- Whether a nonpersistent message is retrieved outside sync point, allowing fast messaging
- z/OS On IBM MQ for z/OS, whether the message retrieved is marked as skipping backout (see “Skipping backout” on page 771)
- 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
- z/OS On IBM 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
- z/OS On IBM 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 773)
- The order in which messages and segments are retrieved from a queue z/OS (except for IBM MQ for z/OS)
- Whether complete, logical messages only are retrievable z/OS (except for IBM MQ for z/OS)
- Whether messages in a group can be retrieved only when *all* messages in the group are available
- Whether segments in a logical message can be retrieved only when *all* segments in the logical message are available z/OS (except for IBM MQ for z/OS)

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. z/OS Also, in IBM 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 sync points is determined by the platform:

Platform	Under sync point control
IBM i	No
AIX and Linux systems	No
z/OS z/OS z/OS	Yes
Windows systems	No

- z/OS On IBM 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.

z/OS On IBM 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 about these options, see "Waiting for messages" on page 769 z/OS and "Signaling" on page 770 .

z/OS ***Signal1***
Signal1 is supported only on IBM MQ for z/OS.

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 IBM MQ on all other platforms, the *Signal1* field is reserved and its value is not significant.

z/OS For more information, see "Signaling" on page 770.

Signal2
The *Signal2* field is reserved on all platforms and its value is not significant.

z/OS For more information, see "Signaling" on page 770.

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.

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.

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 large 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 large 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 764.

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 25). 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 758.

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.

z/OS On IBM 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 763.

Logical and physical ordering

Within each priority level, messages on queues can occur 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, next 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 42. 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 61 on page 748:

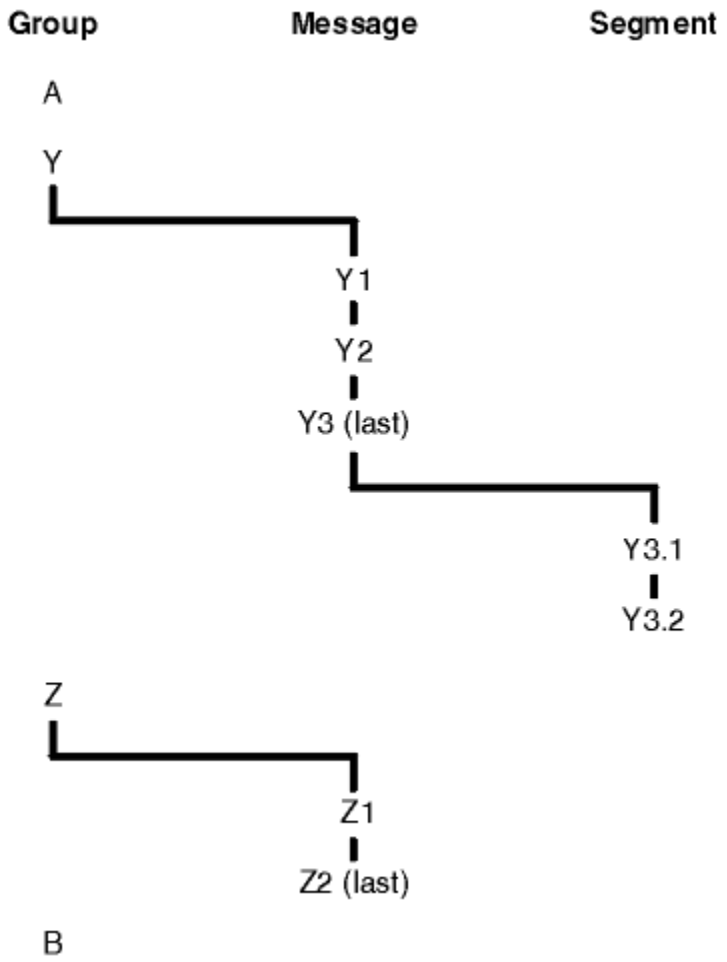


Figure 61. Logical order on a queue

These messages would occur 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. 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 62 on page 749:

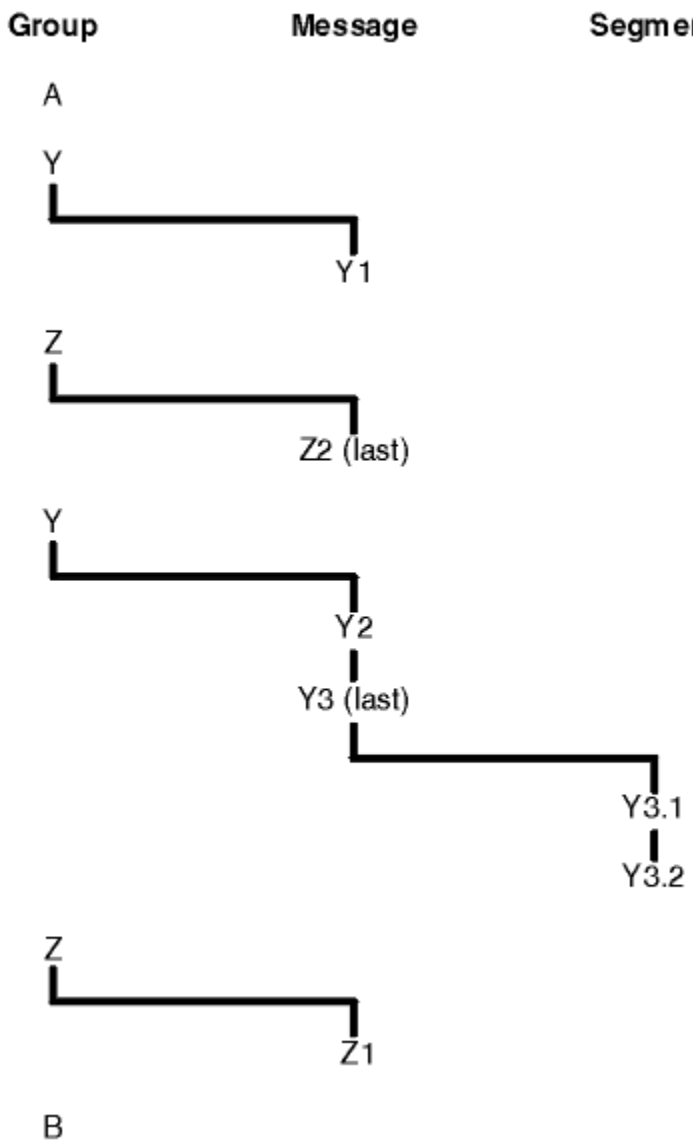


Figure 62. Physical order on a queue

These messages occur 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: z/OS On IBM 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 order 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 MQGMO_LOGICAL_ORDER. 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: Do not 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 MQGET, MQMD, and Rules for validating MQI options.

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.

MQPMO_LOGICAL_ORDER

The MQPMO option tells the queue manager how the application puts messages in groups and segments of logical messages. It can be specified only on the MQPUT call; it is not valid on the MQPUT1 call.

If MQPMO_LOGICAL_ORDER is specified, it indicates that the application uses successive MQPUT calls to:

1. Put the segments in each logical message in the order of increasing segment offset, starting from 0, with no gaps.
2. Put all the segments in one logical message before putting the segments in the next logical message.
3. Put the logical messages in each message group in the order of increasing message sequence number, starting from 1, with no gaps. IBM MQ increments the message sequence number automatically.
4. Put all the logical messages in one message group before putting logical messages in the next message group.

Because the application has told the queue manager how it puts messages in groups and segments of logical messages, the application does not have to maintain and update the group and segment information about each MQPUT call, because the queue manager maintains and updates this information. Specifically, it means that the application does not need to set the *GroupId*, *MsgSeqNumber*, and *Offset* fields in MQMD, because the queue manager sets these fields to the

appropriate values. The application must only set the *MsgFlags* field in MQMD, to indicate when messages belong to groups or are segments of logical messages, and to indicate the last message in a group or last segment of a logical message.

After a message group or logical message has been started, subsequent MQPUT calls must specify the appropriate MQMF_* flags in *MsgFlags* in MQMD. If the application tries to put a message that is not in a group when there is an unterminated message group, or put a message that is not a segment when there is an unterminated logical message, the call fails with reason code MQRC_INCOMPLETE_GROUP or MQRC_INCOMPLETE_MSG, as appropriate. However, the queue manager retains the information about the current message group or current logical message, and the application can terminate them by sending a message (possibly with no application message data) specifying MQMF_LAST_MSG_IN_GROUP or MQMF_LAST_SEGMENT as appropriate, before reissuing the MQPUT call to put the message that is not in the group or not a segment.

Figure 62 on page 749 shows the combinations of options and flags that are valid, and the values of the *GroupId*, *MsgSeqNumber*, and *Offset* fields that the queue manager uses in each case. Combinations of options and flags that are not shown in the table are not valid. The columns in the table have the following meanings; Either means Yes or No:

LOG ORD
Whether the MQPMO_LOGICAL_ORDER option is specified on the call.

MIG
Whether the MQMF_MSG_IN_GROUP or MQMF_LAST_MSG_IN_GROUP option is specified on the call.

SEG
Whether the MQMF_SEGMENT or MQMF_LAST_SEGMENT option is specified on the call.

SEG OK
Whether the MQMF_SEGMENTATION_ALLOWED option is specified on the call.

Cur grp
Whether a current message group exists before the call.

Cur log msg
Whether a current logical message exists before the call.

Other columns
Show the values that the queue manager uses. Previous denotes the value used for the field in the previous message for the queue handle.

Table 116. MQPUT options relating to messages in groups and segments of logical messages

Options you specify	Options you specify	Options you specify	Options you specify	Group and log-msg status before call	Group and log-msg status before call	Values the queue manager uses	Values the queue manager uses	Values the queue manager uses
LOG ORD	MIG	SEG	SEG OK	Cur grp	Cur log msg	**GroupI**d	**MsgSeqNumber**	**Offset**
Yes	No	No	No	No	No	MQGI_NONE	1	0
Yes	No	No	Yes	No	No	New group id	1	0
Yes	No	Yes	Either	No	No	New group id	1	0
Yes	No	Yes	Either	No	Yes	Previous group id	1	Previous offset + previous segment length

Table 116. MQPUT options relating to messages in groups and segments of logical messages (continued)

Options you specify	Options you specify	Options you specify	Options you specify	Group and log-msg status before call	Group and log-msg status before call	Values the queue manager uses	Values the queue manager uses	Values the queue manager uses
Yes	Yes	Either	Either	No	No	New group id	1	0
Yes	Yes	Either	Either	Yes	No	Previous group id	Previous sequence number + 1	0
Yes	Yes	Yes	Either	Yes	Yes	Previous group id	Previous sequence number	Previous offset + previous segment length
No	No	No	No	Either	Either	MQGI_NONE	1	0
No	No	No	Yes	Either	Either	New group ID if MQGI_NONE, else value in field	1	0
No	No	Yes	Either	Either	Either	New group ID if MQGI_NONE, else value in field	1	Value in field
No	Yes	No	Either	Either	Either	New group ID if MQGI_NONE, else value in field	Value in field	0
No	Yes	Yes	Either	Either	Either	New group ID if MQGI_NONE, else value in field	Value in field	Value in field

Note:

- MQPMO_LOGICAL_ORDER is not valid on the MQPUT1 call.
- For the *MsgId* field, the queue manager generates a new message identifier if MQPMO_NEW_MSG_ID or MQMI_NONE is specified, and uses the value in the field otherwise.
- For the *CorrelId* field, the queue manager generates a new correlation identifier if MQPMO_NEW_CORREL_ID is specified, and uses the value in the field otherwise.

When you specify MQPMO_LOGICAL_ORDER, the queue manager requires that all messages in a group and segments in a logical message are put with the same value in the *Persistence* field in MQMD, that is, all must be persistent, or all must be nonpersistent. If this condition is not satisfied, the MQPUT call fails with reason code MQRC_INCONSISTENT_PERSISTENCE.

The MQPMO_LOGICAL_ORDER option affects units of work as follows:

- If the first physical message in a group or logical message is put within a unit of work, all the other physical messages in the group or logical message must be put within a unit of work, if the same queue handle is used. However, they do not need to be put within the same unit of work, allowing a message group or logical message that consists of many physical messages to be split across two or more consecutive units of work for the queue handle.
- If the first physical message in a group or logical message is not put within a unit of work, none of the other physical messages in the group or logical message can be put within a unit of work, if the same queue handle is used.

If these conditions are not satisfied, the MQPUT call fails with reason code MQRC_INCONSISTENT_UOW.

When MQPMO_LOGICAL_ORDER is specified, the MQMD supplied on the MQPUT call must not be less than MQMD_VERSION_2. If this condition is not satisfied, the call fails with reason code MQRC_WRONG_MD_VERSION.

If MQPMO_LOGICAL_ORDER is not specified, messages in groups and segments of logical messages can be put in any order, and it is not necessary to put complete message groups or complete logical messages. It is the responsibility of the application to ensure that the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields have appropriate values.

Use this technique to restart a message group or logical message in the middle, after a system failure has occurred. When the system restarts, the application can set the *GroupId*, *MsgSeqNumber*, *Offset*, *MsgFlags*, and *Persistence* fields to the appropriate values, and then issue the MQPUT call with MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT set as required, but without specifying MQPMO_LOGICAL_ORDER. If this call is successful, the queue manager retains the group and segment information, and subsequent MQPUT calls using that queue handle can specify MQPMO_LOGICAL_ORDER as normal.

The group and segment information that the queue manager retains for the MQPUT call is separate from the group and segment information that it retains for the MQGET call.

For any given queue handle, the application can mix MQPUT calls which specify MQPMO_LOGICAL_ORDER with MQPUT calls that do not, but note the following points:

- If MQPMO_LOGICAL_ORDER is not specified, each successful MQPUT call causes the queue manager to set the group and segment information for the queue handle to the values specified by the application, replacing the existing group and segment information retained by the queue manager for the queue handle.
- If MQPMO_LOGICAL_ORDER is not specified, the call does not fail if there is a current message group or logical message; the call might succeed with an MQCC_WARNING completion code. Table 117 on page 753 shows the various cases that can arise. In these cases, if the completion code is not MQCC_OK, the reason code is one of the following (as appropriate):
  - MQRC_INCOMPLETE_GROUP
  - MQRC_INCOMPLETE_MSG
  - MQRC_INCONSISTENT_PERSISTENCE
  - MQRC_INCONSISTENT_UOW

  Note: The queue manager does not check the group and segment information for the MQPUT1 call.

Table 117. Outcome when MQPUT or MQCLOSE call is not consistent with group and segment information

Current call is	Previous call was MQPUT with MQPMO_LOGICAL_ORDER	Previous call was MQPUT without MQPMO_LOGICAL_ORDER
MQPUT with MQPMO_LOGICAL_ORDER	MQCC_FAILED	MQCC_FAILED
MQPUT without MQPMO_LOGICAL_ORDER	MQCC_WARNING	MQCC_OK
MQCLOSE with an unterminated group or logical message	MQCC_WARNING	MQCC_OK

For applications that put messages and segments in logical order, specify MQPMO_LOGICAL_ORDER, as it is the simplest option to use. This option relieves the application of the need to manage the group and segment information, because the queue manager manages that information. However, specialized applications might need more control than that provided by the MQPMO_LOGICAL_ORDER option, which can be achieved by not specifying that option; if you do so, you must ensure that the

GroupId, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD are set correctly, before each MQPUT or MQPUT1 call.

For example, an application that wants to forward physical messages that it receives, without regard for whether those messages are in groups or segments of logical messages, must not specify MQPMO_LOGICAL_ORDER, for two reasons:

- If the messages are retrieved and put in order, specifying MQPMO_LOGICAL_ORDER assigns a new group identifier to the messages, which might make it difficult or impossible for the originator of the messages to correlate any reply or report messages that result from the message group.
- In a complex network with multiple paths between sending and receiving queue managers, the physical messages might arrive out of order. By not specifying MQPMO_LOGICAL_ORDER and MQGMO_LOGICAL_ORDER on the MQGET call, the forwarding application can retrieve and forward each physical message as soon as it arrives, without waiting for the next one in logical order to arrive.

Applications that generate report messages for messages in groups or segments of logical messages must also not specify MQPMO_LOGICAL_ORDER when putting the report message.

MQPMO_LOGICAL_ORDER can be specified with any of the other MQPMO_* options.

Putting Logically Ordered Groups to a Clustered Queue (MQOO_BIND_ON_GROUP)

The MQOO_BIND_ON_OPEN option ensures that all messages from this application, and therefore all groups, are routed to a single instance. This has the drawback that the application traffic is not load balanced across multiple instances of a cluster queue. In order to enable workload balancing while keeping groups of messages intact, you must set the following options:

- The MQPUT call must specify MQPMO_LOGICAL_ORDER
- The MQOPEN call must specify one of the following two options:
  - MQOO_BIND_ON_GROUP
  - MQOO_BIND_AS_Q_DEF, and the queue definition must specify DEFBIND(GROUP)

Workload balancing is then driven *between groups* of messages without requiring an MQCLOSE and MQOPEN of the queue. *Between groups* means that MQMF_MSG_IN_GROUP is set in the MQMD(v2) or MQMDE, and there is no partially complete group in progress. When a group is in progress, the resolved queue manager and queue name in the object handle are reused.

If the previous message was MQPMO_LOGICAL_ORDER and/or MQMF_MSG_IN_GROUP was set but the current message is not part of the group, then the PUT call fails with MQRC_INCOMPLETE_GROUP.

If an individual MQPUT does not specify MQPMO_LOGICAL_ORDER, and no current group is active, then workload balancing is driven for that message (as if the MQOPEN call has specified MQOO_BIND_NOT_FIXED).

No reallocation takes place for messages bound to a destination using MQOO_BIND_ON_GROUP. For more information on reallocation, see “Message groups” on page 42.

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 specifies the MQGMO_ALL_MSGS_AVAILABLE option for the first message in the group. This ensures that processing does not start until all the messages within the group have arrived. The MQGMO_ALL_MSGS_AVAILABLE option is ignored for subsequent messages within the group.

When the first logical message of the group is retrieved, you can 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 766 and "Putting and getting a group that spans units of work" on page 755.

Attention: When using publish/subscribe to send messages to a topic (or putting messages to a topic alias) message grouping and segmentation is not permitted.

Because subscriptions can be created and removed independently of publication activity, it cannot be assured that a subscriber would receive a full message group or all segments of a message; see RC2417: MQRC_MSG_NOT_ALLOWED_IN_GROUP.

For information on allowing an application to request that a group of messages are all allocated to the same destination instance for cluster queues, see DefBind.

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. In order to realize the benefits, use several units of work for each group of messages. For recovery reasons, you must retrieve each message within a unit of work.

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 118 on page 758 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 118. 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 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 119 on page 758 shows which message is retrieved for the possible settings of these fields.

Table 119. 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 747 and Options.

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 must 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 between MQGETs.

z/OS On IBM 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 763.

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 30.

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 a 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.

When you call MQOPEN with MQOO_READ_AHEAD, the IBM MQ client only enables read-ahead if certain conditions are met. These conditions include:

- The client application must be compiled and linked against the threaded IBM MQ MQI client libraries.
- The client channel must be using TCP/IP protocol
- The channel must have a non-zero SharingConversations (SHARECNV) setting in both the client and server channel definitions.

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.

If a client application is restarted, messages in the read ahead buffer can be lost. Conversely, a message that has been moved to a read ahead buffer could then be deleted from the underlying queue; this does not result in it being removed from the buffer, so an MQGET call using read ahead can return a message that no longer exists.

Read ahead is only performed for client bindings. The attribute is ignored for all other bindings.

Read ahead has no effect 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 IBM 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.

Related concepts

"Tuning performance for nonpersistent messages on AIX" on page 762
If you are using AIX V5.3 or later, consider setting your tuning parameter to use full performance for nonpersistent messages.

Related tasks

"Enabling and disabling read ahead" on page 761
By default read ahead is disabled. You can enable read ahead at queue or application level.

Related reference

"MQGET options and read ahead" on page 760
Not all MQGET options are supported when read ahead is enabled; some options are required to be consistent between MQGET calls.

MQGET options and read ahead

Not all MQGET options are supported when read ahead is enabled; some options are required to be consistent between MQGET calls.

When you call MQOPEN with MQOO_READ_AHEAD, the IBM MQ client only enables read-ahead if certain conditions are met. These conditions include:

- The client application must be compiled and linked against the threaded IBM MQ MQI client libraries.
- The client channel must be using TCP/IP protocol
- The channel must have a non-zero SharingConversations (SHARECNV) setting in both the client and server channel definitions.

The following table indicates which options are supported for use with read ahead and whether they can be altered between MQGET calls.

Table 120. MQGET options and read ahead

MQGET values and options	Permitted when read ahead is enabled and can be altered between MQGET calls [5]	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	

Table 120. MQGET options and read ahead (continued)

MQGET values and options	Permitted when read ahead is enabled and can be altered between MQGET calls [5]	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 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 is returned.
2. If these options are specified on the first MQGET call then read ahead is disabled. If these options are specified on a subsequent MQGET call a reason code MQRC_OPTIONS_ERROR is returned.
3. If a client application alters MsgId and CorrelId values between MQGET calls, messages with the previous values might already have 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.
5. When read ahead is enabled, the first MQGET determines whether messages are to be browsed or got from a queue. If the client application then uses MQGET with changed options, such as attempting to browse following an initial get, or attempting to get following an initial browse, an MQRC_OPTIONS_CHANGED reason code is returned.

If a client alters its selection criteria between MQGET calls, messages being stored in the read ahead buffer that match the initial selection criteria are not consumed by the client application, and remain stranded in the client read ahead buffer. In situations where the client read ahead buffer contains many stranded messages, the benefits associated with read ahead are lost and a separate request to the server is 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 shows read ahead as being inhibited.

If, because of these restrictions on MQGET, you decide 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.

About this task

When you call MQOPEN with MQOO_READ_AHEAD, the IBM MQ client only enables read-ahead if certain conditions are met. These conditions include:

- The client application must be compiled and linked against the threaded IBM MQ MQI client libraries.

- The client channel must be using TCP/IP protocol
- The channel must have a non-zero SharingConversations (SHARECNV) setting in both the client and server channel definitions.

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 is not 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 returns with MQCC_WARNING.

Tuning performance for nonpersistent messages on AIX
If you are using AIX V5.3 or later, consider setting your tuning parameter to use full performance for nonpersistent messages.

To set the tuning parameter so that it takes effect immediately, issue the following command as a root user:

```
/usr/sbin/ioo -o j2_nPagesPerWriteBehindCluster=0
```

To set the tuning parameter so that it takes effect immediately and persists over reboots, issue the following command as a root user:

```
/usr/sbin/ioo -p -o j2_nPagesPerWriteBehindCluster=0
```

Normally, nonpersistent messages are kept only in memory, but there are circumstances where AIX can schedule nonpersistent messages to be written to disk. Messages scheduled to be written to disk are unavailable for MQGET until the disk write completes. The suggested tuning command varies this threshold; instead of scheduling messages to be written to disk when 16 kilobytes of data are queued, the write-to-disk occurs only when real storage on the machine becomes close to full. This is a global alteration and might affect other software components.

On AIX, when using multithreaded applications and especially when running on machines with multiple processors, we strongly recommend setting AIXTHREAD_SCOPE=S in the mqm ID `.profile` or setting AIXTHREAD_SCOPE=S in the environment before starting the application, for better performance and more solid scheduling. For example:

```
export AIXTHREAD_SCOPE=S
```

Setting AIXTHREAD_SCOPE=S means that user threads created with default attributes are placed into system-wide contention scope. If a user thread is created with system-wide contention scope, it is bound

to a kernel thread and it is scheduled by the kernel. The underlying kernel thread is not shared with any other user thread.

File descriptors

When running a multi-threaded process such as the agent process, you might reach the soft limit for file descriptors. This limit gives you the IBM MQ reason code `MQRC_UNEXPECTED_ERROR (2195)` and, if there are enough file descriptors, an IBM MQ FFST™ file.

To avoid this problem, you can increase the process limit for the number of file descriptors. To do so, alter the `nofiles` attribute in `/etc/security/limits` to 10,000 for the mqm user ID or in the default stanza.

System Resource Limits

Set the system resource limit for data segment and stack segment to unlimited using the following commands in a command prompt:

```
ulimit -d unlimited
ulimit -s unlimited
```

Type of index

The queue attribute, *IndexType*, specifies the type of index that the queue manager maintains to increase the speed of MQGET operations on the queue.

Note: Supported only on IBM MQ for z/OS.

You have five options:

Value	Description
NONE	No index is maintained. Use this when retrieving messages sequentially (see "Priority" on page 747).
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 747).
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 758).
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 758).

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 IndexType. For further information on the **IndexType** attribute, see “Design and performance considerations for z/OS applications” on page 62.

Handling messages greater than 4 MB long

Messages can be too large for the application, queue, or queue manager. Depending on the environment, IBM 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 IBM MQ systems at V6 or later. Set this value to reflect the size of the messages using the queue. On IBM MQ for Multiplatforms, 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.

Multi On IBM MQ for Multiplatforms, you can set both these attributes manually. You can set the queue manager attribute value in the range 32768 bytes through 100 MB.

Attention: z/OS On IBM MQ for z/OS the **MaxMsgLength** queue manager attribute is hard coded at 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.

Multi *Message segmentation*

Use this information to learn about segmenting messages. This feature is not supported on IBM MQ for z/OS or by applications using IBM MQ classes for JMS.

Increasing the maximum message length as explained in topic “Increasing the maximum message length” on page 764 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 42.

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. Always consider using this technique to reduce the possibility of incomplete groups being present in the network. Single-phase commit by the queue manager is assumed, but 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).

Attention: When using publish/subscribe to send messages to a topic (or putting messages to a topic alias) message grouping and segmentation is not permitted.

Because subscriptions can be created and removed independently of publication activity, it cannot be assured that a subscriber would receive a full message group or all segments of a message; see RC2417: MQRC_MSG_NOT_ALLOWED_IN_GROUP.

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 755.

You cannot, however, put or get segmented messages in a global unit of work.

Multi *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)
   MD.MsgFlags = MQMF_SEGMENTATION_ALLOWED
   MD.Version = MQMD_VERSION_2
   memcpy(MD.GroupId, MQGI_NONE, MQ_GROUP_ID_LENGTH)
MQPUT
```

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
```

In this simplest scenario, the application must reset the GroupId field to MQGI_NONE before the MQPUT call, so that the queue manager can generate a unique group identifier for each message. If this is not done, unrelated messages can have the same group identifier, which might subsequently lead to incorrect processing.

The application buffer must be large enough to contain the reassembled message (unless you include the MQGMO_ACCEPT_TRUNCATED_MSG option).

If the MAXMSGLEN attribute of a queue is to be modified to accommodate message segmentation, then consider:

- The minimum message segment supported on a local queue is 16 bytes.
- For a transmission queue, MAXMSGLEN must also include the space required for headers. Consider using a value at least 4000 bytes larger than the maximum expected length of user data in any message segment that could be put on a transmission queue.

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.

Multi *Application segmentation*

Application segmentation is used when queue manager segmentation is not adequate, or when applications require data conversion with specific segment boundaries.

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 must 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.

After 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 occurs 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
```

Multi *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_MSGS_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.

Multi *Reference messages and large object transfers*

Reference messages allow a large object to be transferred from one node to another without storing the object on IBM MQ queues at either the source or the destination nodes. This is of particular benefit when the data exists in another form, for example, for mail applications.

To enable this transfer method, you specify a message exit at both ends of a channel. For information about how to do this, see “Channel message exit programs” on page 943.

IBM MQ defines the format of a reference message header (MQRMH). See MQRMH for a description of this. This is recognized with 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 receives 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 points:

- The MCA and the message exit run under an IBM 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 IBM 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 an IBM 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 MQRMH).
- 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 node might be the final node for some of the destinations in the list, but an intermediate node for others.
- Bulk data is not typically 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 MQRMH and MQMD 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 IBM 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 are 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 "Using the sample programs on Multiplatforms" on page 1018.

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 MQGMO.

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.
- z/OS 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.

z/OS If you want your application to wait on more than one queue, use the signal facility of IBM MQ for z/OS (see “Signaling” on page 770). For more information about the circumstances in which these actions occur, see MQGMO.

Signaling

Signaling is supported only on IBM 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 like the *get with wait* function described in topic “Waiting for messages” on page 769 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 IBM 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 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. IBM 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** parameter is set to MQCC_WARNING and the **Reason** parameter is set to MQRC_SIGNAL_REQUEST_ACCEPTED.

For information about what IBM MQ can return to your application when it makes an MQGET call using signaling, see MQGET.

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 z/OS 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 MQGMO - Get-message options.
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 MQGMO - Get-message options.
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 Signal1.

z/OS *Skipping backout*

You can prevent an application program from entering an *MQGET-error-backout* loop by specifying the **MQGMO_MARK_SKIP_BACKOUT** option on the MQGET call.

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 must 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 an IBM 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 63 on page 772 illustrates a typical sequence of steps that an application program might contain when an MQGET request is required to skip backout.

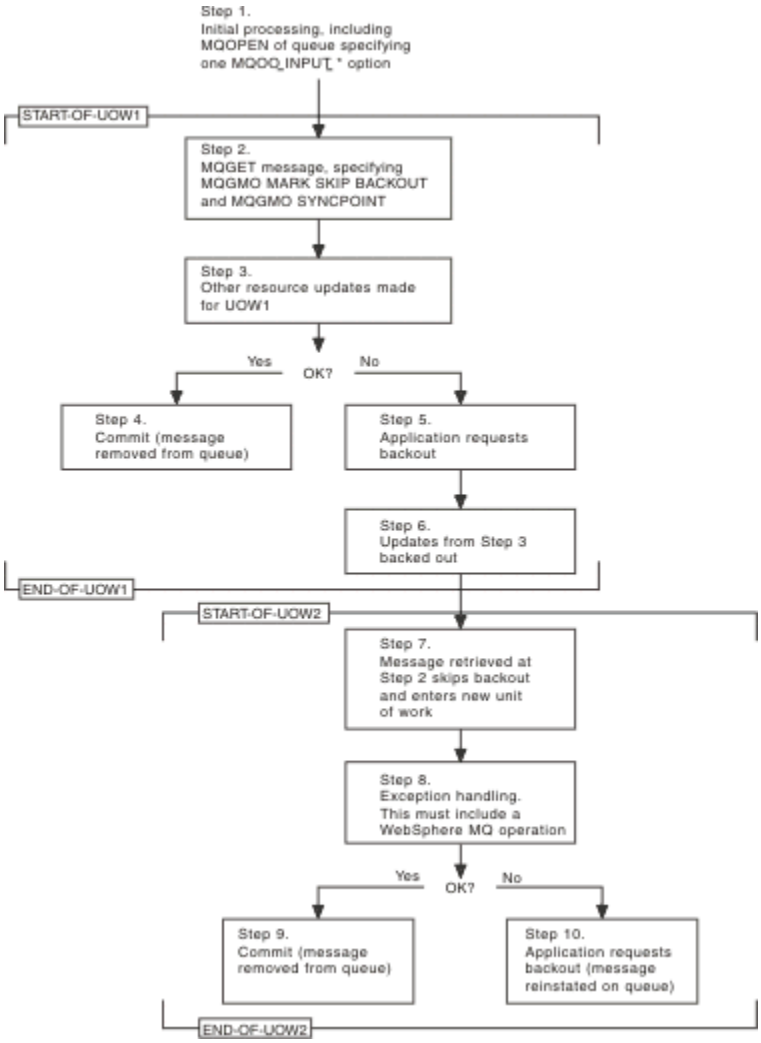


Figure 63. Skipping backout using MQGMO_MARK_SKIP_BACKOUT

The steps in Figure 63 on page 772 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 63 on page 772 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 when *all* the following statements 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

CodedCharSetId and Encoding 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 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 947. See also National languages and Machine encodings 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 IBM 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 IBM i system remotely from an ASCII platform using RUNMQSC.

See Data conversion for further information about converting EBCDIC-format data to ASCII format.

Browsing messages on a queue

Use this information to find out about browsing messages on a queue using the MQGET call.

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 746.)

- "Queues in FIFO (first in, first out) sequence" on page 775
- "Queues in priority sequence" on page 775
- "Uncommitted messages" on page 775
- "Change to queue sequence" on page 776
- z/OS "Using the queue index" on page 776

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 later, take the default priority of the queue.

Using the queue index

z/OS

On IBM 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 uses the index to browse when certain forms of browse are used.

Any of the following forms of browse are used when an indexed queue contains only messages of single priority:

1. If the queue is indexed by MSGID, 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, 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, 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 does not add significant extra processing 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 747 explains 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 777, 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 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 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 MQOPEN. 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 publish/subscribe applications

Start writing publish/subscribe IBM MQ applications.

For an overview of publish/subscribe concepts, see Publish/subscribe messaging.

See the following topics for information on writing different types of publish/subscribe applications:

- “Writing publisher applications” on page 781
- “Writing subscriber applications” on page 787
- “Publish/subscribe lifecycles” on page 803
- “Publish/subscribe message properties” on page 808
- “Message ordering” on page 809
- “Intercepting publications” on page 810
- “Publishing options” on page 818
- “Subscription options” on page 818

Related concepts

“Application development concepts” on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

“Developing applications for IBM MQ” on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

“Design considerations for IBM MQ applications” on page 47
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 IBM MQ.

“Writing a procedural application for queuing” on page 696

Use this information to learn about writing queuing applications, connecting to and disconnecting from a queue manager, publish/subscribe, and opening and closing objects.

"Writing client procedural applications" on page 879
What you need to know to write client applications on IBM MQ using a procedural language.

"Building a procedural application" on page 962
You can write an IBM MQ application in one of several procedural languages, and run the application on several different platforms.

"Handling procedural program errors" on page 999
This information explains errors associated with your applications MQI calls either when it makes a call, or when its message is delivered to its final destination.

Related tasks

"Using the IBM MQ sample procedural programs" on page 1018
These sample programs are written in procedural languages, and demonstrate typical uses of the Message Queue Interface (MQI). IBM MQ programs on different platforms.

Writing publisher applications

Get started with writing publisher applications by studying two examples. The first is modeled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Writing a simple IBM MQ publisher application is just like writing an IBM MQ point to point application that puts messages to a queue (Table 121 on page 781). The difference is you MQPUT messages to a topic, not to a queue.

Table 121. Point to point versus publish/subscribe IBM MQ program pattern.

Step	Point to point MQ Call	Publish MQ Call
Connect to a queue manager	MQCONN	MQCONN
Open queue	MQOPEN	
Open topic		MQOPEN
Put message(s)	MQPUT	MQPUT
Close topic		MQCLOSE
Close queue	MQCLOSE	
Disconnect from queue manager	MQDISC	MQDISC

To make that concrete, there are two examples of applications to publish stock prices. In the first example ("Example 1: Publisher to a fixed topic" on page 782), that is modeled closely on putting messages to a queue, the administrator creates a topic definition in a similar way to creating a queue. The programmer codes MQPUT to write messages to the topic instead of writing them to a queue. In the second example ("Example 2: Publisher to a variable topic" on page 784), the pattern of interaction of the program with IBM MQ is similar. The difference is the programmer provides the topic to which the message is written, rather than the administrator. In practice this typically means that the topic string is content defined, or provided by another source, such as human input through a browser.

Related concepts

"Writing subscriber applications" on page 787
Get started with writing subscriber applications by studying three examples: an IBM MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

Related reference

DEFINE TOPIC
DISPLAYTOPIC

DISPLAYTPSTATUS

Example 1: Publisher to a fixed topic
An IBM MQ program to illustrate publishing to an administratively defined topic.

Note: The compact coding style is intended for readability not production use.

See the output in Figure 65 on page 783

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
int main(int argc, char **argv)
{
    char     topicNameDefault[]   = "IBMSTOCKPRICE";
    char     publicationDefault[] = "129";
    MQCHAR48 qmName = "";

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle                    */
    MQHOBJ   Hobj = MQHO_NONE;            /* object handle sub queue              */
    MQLONG   CompCode = MQCC_OK;          /* completion code                      */
    MQLONG   Reason = MQRC_NONE;          /* reason code                          */
    MQOD     td = {MQOD_DEFAULT};         /* Object descriptor                    */
    MQMD     md = {MQMD_DEFAULT};         /* Message Descriptor                   */
    MQPMO   pmo = {MQPMO_DEFAULT};        /* put message options                  */
    MQCHAR   resTopicStr[151];            /* Returned vale of topic string        */
    char *   topicName   = topicNameDefault;
    char *   publication = publicationDefault;
    memset   (resTopicStr, 0 , sizeof(resTopicStr));

    switch(argc){                         /* replace defaults with args if provided */
        default:
            publication = argv[2];
        case(2):
            topicName = argv[1];
        case(1):
            printf("Optional parameters:  TopicObject Publication\n");
    }
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        td.ObjectType = MQOT_TOPIC;        /* Object is a topic                    */
        td.Version = MQOD_VERSION_4;       /* Descriptor needs to be V4            */
        strncpy(td.ObjectName, topicName, MQ_TOPIC_NAME_LENGTH);
        td.ResObjectString.VSPtr = resTopicStr;
        td.ResObjectString.VSBufSize = sizeof(resTopicStr)-1;
        MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        pmo.Options = MQPMO_FAIL_IF_QUIESCING | MQPMO_RETAIN;
        MQPUT(Hconn, Hobj, &md, &pmo, (MQLONG)strlen(publication)+1, publication, &CompCode,
&Reason);
        if (CompCode != MQCC_OK) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    if (CompCode == MQCC_OK)
        printf("Published \"%s\" using topic \"%s\" to topic string \"%s\"\n",
               publication, td.ObjectName, resTopicStr);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
```

Figure 64. Simple IBM MQ publisher to a fixed topic.

```
X:\Publish1\Debug>PublishStock
Optional parameters:  TopicObject Publication
Published "129" using topic "IBMSTOCKPRICE" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0

X:\Publish1\Debug>PublishStock IBMSTOCKPRICE 155
Optional parameters:  TopicObject Publication
Published "155" using topic "IBMSTOCKPRICE" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0
```

Figure 65. Sample output from first publisher example

The following selected lines of code illustrate aspects of writing a publisher application for IBM MQ.

char topicNameDefault[] = "IBMSTOCKPRICE";
A default topic name is defined in the program. You can override it by providing the name of a different topic object as the first argument to the program.

MQCHAR resTopicStr[151];
`resTopicStr` is pointed at by `td.ResObjectString.VSPtr` and is used by `MQOPEN` to return the resolved topic string. Make the length of `resTopicStr` one larger than the length passed in `td.ResObjectString.VSBufSize` to give space for null termination.

memset (resTopicStr, 0, sizeof(resTopicStr));
Initialize `resTopicStr` to nulls to ensure the resolved topic string returned in an `MQCHARV` is null terminated.

td.ObjectType = MQOT_TOPIC
There is a new type of object for publish/subscribe: the *topic object*.

td.Version = MQOD_VERSION_4;
To use the new type of object, you must use at least *version 4* of the object descriptor.

strncpy(td.ObjectName, topicName, MQ_OBJECT_NAME_LENGTH);
The `topicName` is the name of a topic object, sometimes called an administrative topic object. In the example the topic object needs to be created beforehand, using IBM MQ Explorer or this MQSC command,

```
DEFINE TOPIC(IBMSTOCKPRICE) TOPICSTR(NYSE/IBM/PRICE) REPLACE;
```

td.ResObjectString.VSPtr = resTopicStr;
The resolved topic string is echoed in the final `printf` in the program. Set up the `MQCHARV` `ResObjectString` structure for IBM MQ to return the resolved string back to the program.

MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
Open the topic for output; just like opening a queue for output.

pmo.Options = MQPMO_FAIL_IF_QUIESCING | MQPMO_RETAIN;
You want new subscribers to be able receive the publication, and by specifying `MQPMO_RETAIN` in the publisher, when you start a subscriber it receives the latest publication, published before the subscriber started, as its first matching publication. The alternative is to provide subscribers with publications published only after the subscriber started. Additionally a subscriber has the option to decline to receive a retained publication by specifying `MQSO_NEW_PUBLICATIONS_ONLY` in its subscription.

MQPUT(Hconn, Hobj, &md, &pmo, (MQLONG)strlen(publication)+1, publication, &CompCode, &Reason);
Add 1 to the length of the string passed to `MQPUT` to pass the null termination character to IBM MQ as part of the message buffer.

What does the first example demonstrate? The example imitates as closely as possible the tried and tested traditional pattern for writing point to point IBM MQ programs. An important feature of the IBM MQ programming pattern is that the programmer is not concerned where messages are sent. The task of the programmer is to connect to a queue manager, and pass it the messages that are to be distributed to recipients. In the point-to-point paradigm, the programmer opens a queue (probably an alias queue) that

the administrator has configured. The alias queue routes messages to a target queue, either on the local queue manager, or to a remote queue manager. While the messages are waiting to be delivered, they are stored on queues somewhere between the source and the destination.

In the publish/subscribe pattern, instead of opening a queue, the programmer opens a topic. In our example, the topic is associated with a topic string by an administrator. The queue manager forwards the publication, using queues, to local or remote subscribers that have subscriptions that match the topic string of the publication. If publications are retained the queue manager keeps the latest copy of the publication, even if it has no subscribers now. The retained publication is available to forward to future subscribers. The publisher application plays no part in selecting or routing the publication to a destination; its task is to create and put publications to the topics defined by the administrator.

This fixed topic example is atypical of many publish/subscribe applications: it is static. It requires an administrator to define the topic strings and change the topics that are published on. Commonly publish/subscribe applications need to know some or all the topic tree. Perhaps topics change frequently, or perhaps although the topics do not change much, the number of topic combinations is large and it is too onerous for an administrator to define a topic node for every topic string that might need to be published on. Perhaps topic strings are not known in advance of publication; a publisher application might use information from the publication content to specify a topic string, or it might have information about topic strings to publish on from another source, such as human input from a browser. To cater for more dynamic styles of publishing, the next example shows how to create topics dynamically, as part of the publisher application.

Topics couple publishers and subscribers together. Designing the rules, or architecture, for naming topics, and organizing them in topic trees is an important step in developing a publish/subscribe solution. Look carefully at the extent to which organization of the topic tree binds of publisher and subscriber programs together, and binds them to the content of the topic tree. Ask yourself the question whether changes in the topic tree affect publisher and subscriber applications, and how you can minimize the effect. Built into the architecture of the IBM MQ publish/subscribe model is the notion of an administrative topic object that provides the root part, or root subtree, of a topic. The topic object gives you the option of defining the root part of the topic tree administratively that simplifies application programming and operations, and consequently improves maintainability. For example, if you are deploying multiple publish/subscribe applications that have isolated topic trees, then by administratively defining the root part of the topic tree, you can guarantee the isolation of topic trees, even if there is no consistency in the topic naming conventions adopted by the different applications.

In practice, publisher applications cover a spectrum from solely using fixed topics, as in this example, and variable topics, as in the next. “Example 2: Publisher to a variable topic” on page 784 also demonstrates combining the use of topics and topic strings.

Related concepts

“Example 2: Publisher to a variable topic” on page 784
A WebSphere MQ program to illustrate publishing to a programmatically defined topic.

“Writing subscriber applications” on page 787
Get started with writing subscriber applications by studying three examples: an IBM MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

Example 2: Publisher to a variable topic
A WebSphere MQ program to illustrate publishing to a programmatically defined topic.

Note: The compact coding style is intended for readability not production use.

See the output in Figure 67 on page 785.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
int main(int argc, char **argv)
{
    char     topicNameDefault[]   = "STOCKS";
    char     topicStringDefault[] = "IBM/PRICE";
    char     publicationDefault[] = "130";
    MQCHAR48 qmName = "";

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle           */
    MQHOBJ   Hobj     = MQHO_NONE;   /* object handle sub queue          */
    MQLONG   CompCode = MQCC_OK;     /* completion code                  */
    MQLONG   Reason   = MQRC_NONE;   /* reason code                      */
    MQOD     td = {MQOD_DEFAULT};    /* Object descriptor                */
    MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor               */
    MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options              */
    MQCHAR   resTopicStr[151];       /* Returned value of topic string */
    char *   topicName   = topicNameDefault;
    char *   topicString = topicStringDefault;
    char *   publication = publicationDefault;
    memset   (resTopicStr, 0 , sizeof(resTopicStr));

    switch(argc){            /* Replace defaults with args if provided */
        default:
            publication = argv[3];
        case(3):
            topicString = argv[2];
        case(2):
            if (strcmp(argv[1],"/"))  /* "/" invalid = No topic object  */
                topicName = argv[1];
            else
                *topicName = '\0';
        case(1):
            printf("Provide parameters:  TopicObject TopicString Publication\n");
    }

    printf("Publish \"%s\" to topic \"%-.48s\" and topic string \"%s\"\n", publication, topicName,
topicString);
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        td.ObjectType = MQOT_TOPIC;    /* Object is a topic             */
        td.Version = MQOD_VERSION_4;   /* Descriptor needs to be V4     */
        strncpy(td.ObjectName, topicName,  MQ_TOPIC_NAME_LENGTH);
        td.ObjectString.VSPtr = topicString;
        td.ObjectString.VSLength = (MQLONG)strlen(topicString);
        td.ResObjectString.VSPtr = resTopicStr;
        td.ResObjectString.VSBufSize = sizeof(resTopicStr)-1;
        MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        pmo.Options = MQPMO_FAIL_IF_QUIESCING | MQPMO_RETAIN;
        MQPUT(Hconn, Hobj, &md, &pmo, (MQLONG)strlen(publication)+1, publication, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    if (CompCode == MQCC_OK)
        printf("Published \"%s\" to topic string \"%s\"\n",  publication, resTopicStr);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
```

Figure 66. Simple IBM MQ publisher to a variable topic.

```
X:\Publish2\Debug>PublishStock
Provide parameters: TopicObject TopicString Publication
Publish "130" to topic "STOCKS" and topic string "IBM/PRICE"
Published "130" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0

X:\Publish2\Debug>PublishStock / NYSE/IBM/PRICE 131
Provide parameters: TopicObject TopicString Publication
Publish "131" to topic "" and topic string "NYSE/IBM/PRICE"
Published "131" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0
```

Figure 67. Sample output from second publisher example

There are a few points to note about this example.

char topicNameDefault[] = "STOCKS";
The default topic name STOCKS defines part of the topic string. You can override this topic name by providing it as the first argument to the program, or eliminate the use of the topic name by supplying / as the first parameter.

char topicString[101] = "IBM/PRICE";
IBM/PRICE is the default topic string. You can override this topic string by providing it as the second argument to the program.

The queue manager combines the topic string provided by the STOCKS topic object, "NYSE", with the topic string provided by the program "IBM/PRICE" and inserts a "/" between the two topic strings. The result is the resolved topic string "NYSE/IBM/PRICE". The resulting topic string is the same as the one defined in the IBMSTOCKPRICE topic object, and has precisely the same effect.

The administrative topic object associated with the resolved topic string is not necessarily the same topic object as passed to MQOPEN by the publisher. IBM MQ uses the tree implicit in the resolved topic string to work out which administrative topic object defines the attributes associated with the publication.

Suppose there are two topic objects A and B, and A defines topic "a", and B defines topic "a/b" (Figure 68 on page 786). If the publisher program refers to topic object A and provides topic string "b", resolving the topic to the topic string "a/b", then the publication inherits its properties from topic object B because the topic matches the topic string "a/b" defined for B.

if (strcmp(argv[1],"/"))
argv[1] is the optionally provided topicName. "/" is invalid as a topic name; here it signifies that there is no topic name, and the topic string is provided entirely by the program. The output in Figure 67 on page 785 shows the whole topic string being supplied dynamically by the program.

strncpy(td.ObjectName, topicName, MQ_OBJECT_NAME_LENGTH);
For the default case, the optional topicName needs to be created beforehand, using IBM MQ Explorer or this MQSC command:

```
DEFINE TOPIC(STOCKS) TOPICSTR(NYSE) REPLACE;
```

td.ObjectString.VSPtr = topicString;
The topic string is a MQCHARV field in the topic descriptor

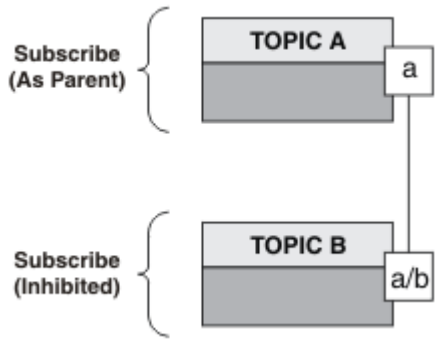


Figure 68. Topic object associations

What does the second example demonstrate? Although the code is very similar to the first example - effectively there are only two lines difference - the result is a significantly different program to the first. The programmer controls the destinations to which publications are sent. In conjunction with minimal administrator input used to design subscriber applications, no topics or queues need to be predefined to route publications from publishers to subscribers.

In the point-to-point messaging paradigm, queues have to be defined before messages are able to flow. For publish/subscribe, they do not, although IBM MQ implements publish/subscribe using its underlying queuing system; the benefits of guaranteed delivery, transactionality and loose coupling associated with messaging and queuing are inherited by publish/subscribe applications.

A designer has to decide whether publisher, and subscriber, programs are to be aware of the underlying topic tree or not, and also whether subscriber programs are aware of queuing or not. Study the subscriber example applications next. They are designed to be used with the publisher examples, typically publishing and subscribing to `NYSE/IBM/PRICE`.

Related concepts

"Example 1: Publisher to a fixed topic" on page 782
An IBM MQ program to illustrate publishing to an administratively defined topic.

"Writing subscriber applications" on page 787
Get started with writing subscriber applications by studying three examples: an IBM MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

Writing subscriber applications

Get started with writing subscriber applications by studying three examples: an IBM MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

In Table 122 on page 787 the three styles of consumer or subscriber are listed, together with the sequences of IBM MQ function calls that characterize them.

1. The first style, MQ Publication Consumer, is identical to a point to point MQ program that only does MQGET. The application has no knowledge that it is consuming publications - it is simply reading messages from a queue. The subscription that causes publications to get routed to the queue is created administratively using IBM MQ Explorer or a command.
2. The second style is the preferred pattern for most subscriber applications. The subscriber application creates the subscription, and then gets publications. The queue management is all performed by the queue manager. This is known as a *managed subscriber*.
3. In the third style, the subscriber application is responsible for specifying the queue that will be used to hold publications, opening and closing that queue and issuing subscriptions to fill the queue with publications. This is known as an *unmanaged subscriber*.

One way to understand these styles is to study the example `C` programs listed in Table 122 on page 787 for each of the styles. The examples are designed to be run in conjunction with the publisher example found in "Writing publisher applications" on page 781.

Table 122. Point to point vs. subscribe IBM MQ program patterns.

Step	MQ message consumer	"Example 1: MQ Publication consumer" on page 788	"Example 2: Managed MQ subscriber" on page 790	"Example 3: Unmanaged MQ subscriber" on page 795
Connect to a queue manager	MQCONN	MQCONN	MQCONN	MQCONN
Open queue	MQOPEN	MQOPEN		MQOPEN
Subscribe			MQSUB	MQSUB
Get message(s)	MQGET	MQGET	MQGET	MQGET
Close queue	MQCLOSE	MQCLOSE	(MQCLOSE)	MQCLOSE
Close subscription			MQCLOSE	MQCLOSE

Table 122. Point to point vs. subscribe IBM MQ program patterns. (continued)				
Step	**MQ message consumer**	**"Example 1: MQ Publication consumer" on page 788**	**"Example 2: Managed MQ subscriber" on page 790**	**"Example 3: Unmanaged MQ subscriber" on page 795**
Disconnect from queue manager	MQDISC	MQDISC	MQDISC	MQDISC

Using MQCLOSE is always optional, either to release resources, pass MQCLOSE options, or just for symmetry with MQOPEN. Since you are unlikely to need to specify the MQCLOSE options when the subscription queue is closed in the Managed MQ subscriber case, and the symmetry argument is not relevant, the subscription queue is not explicitly closed in Example 2: Managed MQ subscriber.

Another way to understand publish/subscribe application patterns is too look at the interactions between the different entities involved. Lifeline, or UML sequence diagrams are a good way to study interactions. Three lifeline examples are described in "Publish/subscribe lifecycles" on page 803.

Example 1: MQ Publication consumer
The MQ Publication consumer is an IBM MQ message consumer that does not subscribe to topics itself.

To create the subscription and publication queue for this example run the following commands, or define the objects using IBM MQ Explorer.

```
DEFINE QLOCAL(STOCKTICKER) REPLACE;
DEFINE SUB(IBMSTOCKPRICESUB) DEST(STOCKTICKER) TOPICOBJ(IBMSTOCKPRICE) REPLACE;
```

The `IBMSTOCKPRICESUB` subscription references the `IBMSTOCK` topic object created for the publisher example and the local queue `STOCKTICKER`. The topic object `IBMSTOCK` defines the topic string that is used in the subscription, `NYSE/IBM/PRICE`. Note that the topic object and the queue used to receive publications need to be defined before the subscription is created.

There are a number of valuable facets to the MQ publication consumer pattern:

1. Multiprocessing: sharing out of the work of reading publications. The publications all go onto the single queue associated with the subscription topic. Multiple consumers can open the queue using `MQOO_INPUT_SHARED`.
2. Centrally managed subscriptions. Applications do not construct their own subscription topics or subscriptions; the administrator is responsible for where publications are sent.
3. Subscription concentration: multiple different subscriptions can be sent to a single queue.
4. Subscription durability: the queue receives all publications whether or not consumers are active.
5. Migration and coexistence: the consumer code works equally well for a point-to-point and a publish/subscribe scenario.

The subscription creates a relationship between the topic string `NYSE/IBM/PRICE` and the queue `STOCKTICKER`. Publications, including any currently retained publication, are forwarded to `STOCKTICKER` from the moment the subscription is created.

An administratively created subscription can be managed or unmanaged. A managed subscription takes effect as soon as it has been created, just like an unmanaged subscription. Not all the pattern facets are available to a managed subscription. See "Example 3: Unmanaged MQ subscriber" on page 795

Note: The compact coding style is intended for readability not production use.

The results are shown in Figure 70 on page 789.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
int main(int argc, char **argv)
{
MQCHAR     publicationBuffer[101];
MQCHAR48 subscriptionQueueDefault = "STOCKTICKER";
MQCHAR48 qmName = "";            /* Use default queue manager  */

MQHCONN Hconn = MQHC_UNUSABLE_HCONN;    /* connection handle      */
MQHOBJ  Hobj = MQHO_NONE;         /* object handle sub queue   */
MQLONG  CompCode = MQCC_OK;        /* completion code       */
MQLONG  Reason = MQRC_NONE;        /* reason code         */
MQLONG  messlen = 0;
MQOD   od = {MQOD_DEFAULT};        /* Unmanaged subscription queue */
MQMD   md = {MQMD_DEFAULT};        /* Message Descriptor      */
MQGMO  gmo = {MQGMO_DEFAULT};       /* Get message options     */
char *  publication=publicationBuffer;
char *  subscriptionQueue = subscriptionQueueDefault;

switch(argc){           /* Replace defaults with args if provided */
default:
subscriptionQueue = argv[1]
case(1):
printf("Optional parameter: subscriptionQueue\n");
}

do {
MQCONN(qmName, &Hconn, &CompCode, &Reason);
if (CompCode != MQCC_OK) break;
strncpy(od.ObjectName, subscriptionQueue, MQ_Q_NAME_LENGTH);
MQOPEN(Hconn, &od, MQOO_INPUT_AS_Q_DEF | MQOO_FAIL_IF_QUIESCING , &Hobj, &CompCode, &Reason);
if (CompCode != MQCC_OK) break;
gmo.Options = MQGMO_WAIT | MQGMO_NO_SYNCPOINT | MQGMO_CONVERT;
gmo.WaitInterval = 10000;
printf("Waiting %d seconds for publications from %s\n", gmo.WaitInterval/1000,
subscriptionQueue);
do {
memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
md.Encoding    = MQENC_NATIVE;
md.CodedCharSetId = MQCCSI_Q_MGR;
memset(publication, 0, sizeof(publicationBuffer));
MQGET(Hconn, Hobj, &md, &gmo, sizeof(publicationBuffer)-1, publication, &messlen,
&CompCode, &Reason);
if (Reason == MQRC_NONE)
printf("Received publication \"%s\"\n", publication);
}
while (CompCode == MQCC_OK);
if (CompCode != MQCC_OK && Reason != MQRC_NO_MSG_AVAILABLE) break;
MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
if (CompCode != MQCC_OK) break;
MQDISC(&Hconn, &CompCode, &Reason);
} while (0);
printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
```

Figure 69. MQ publication consumer.

```
X:\Subscribe1\Debug>Subscribe1
Optional parameter: subscriptionQueue
Waiting 10 seconds for publications from STOCKTICKER
Received publication "129"
Completion code 0 and Return code 0
```

Figure 70. Output from MQ publication consumer

There are a couple of standard IBM MQ *C* language programming tips to be aware of:

memset(publication, 0, sizeof(publicationBuffer));
Ensure the message has a trailing null for easy formatting using `printf`. The publisher example includes the trailing null in the message buffer passed to `MQPUT` by adding 1 to `strlen(publication)`. Setting `MQCHAR` buffers to null is good programming style for IBM MQ C programs that use the buffers to store strings, ensuring a null follows an array of characters that does not completely fill the buffer.

MQGET(Hconn, Hobj, &md, &gmo, sizeof(publicationBuffer)-1, publication, &messlen, &CompCode, &Reason);
Reserve one null at the end of the message buffer to ensure the returned message has trailing null in case `if (messlen == strlen(publication));` is true. This tip complements the preceding one, and ensures that there is at least one null in `publicationBuffer` that is not overwritten by the contents of `publication`.

Related concepts

“Example 2: Managed MQ subscriber” on page 790
The managed MQ subscriber is the preferred pattern for most subscriber applications. A managed subscription is one where IBM MQ handles the subscription and does the registering and de-registering for you. The example requires *no* administrative definition of queues, topics or subscriptions.

“Example 3: Unmanaged MQ subscriber” on page 795
The unmanaged subscriber is an important class of subscriber application. With it, you combine the benefits of publish/subscribe with *control* of queuing and consumption of publications. An unmanaged subscription is where the application is responsible. for specifying the queue where the subscriptions are stored. The example demonstrates different ways of combining subscriptions and queues.

“Writing publisher applications” on page 781
Get started with writing publisher applications by studying two examples. The first is modeled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Example 2: Managed MQ subscriber
The managed MQ subscriber is the preferred pattern for most subscriber applications. A managed subscription is one where IBM MQ handles the subscription and does the registering and de-registering for you. The example requires *no* administrative definition of queues, topics or subscriptions.

This simplest kind of managed subscriber typically uses a *non-durable* subscription. The example focuses on a non-durable subscription. The subscription lasts only as long as the lifetime of the subscription handle from `MQSUB`. Any publications that match the topic string during the lifetime of the subscription are sent to the subscription queue (and possibly a retained publication if the flag `MQSO_NEW_PUBLICATIONS_ONLY` is not set or defaulted, an earlier publication matching the topic string was retained, and the publication was persistent or the queue manager has not terminated, since the publication was created).

You can also use a *durable* subscription with this pattern. Typically if a managed durable subscription is used it is done for reliability reasons, rather than to establish a subscription that, without any errors occurring, would outlive the subscriber. For more information about different life cycles associated with managed, unmanaged, durable and non-durable subscriptions see the related topics section.

Durable subscriptions are often associated with persistent publications, and non-durable subscriptions with non-persistent publications, but there is no necessary relationship between subscription durability and publication persistence. All four combinations of persistence and durability are possible.

For the managed non-durable case considered, the queue manager creates a subscription queue that is purged and deleted when the queue is closed. The publications are removed from the queue when the non-durable subscription is closed.

The valuable facets of the managed non-durable pattern exemplified by this code are as follows:

1. On demand subscription: the subscription topic string is dynamic. It is provided by the application when it runs.
2. Self managing queue: the subscription queue is self defining and managing.

3. Self managing subscription lifecycle: *non-durable* subscriptions only exist for the duration of the subscriber application.
   - If you define a *durable* managed subscription, then it results in a permanent subscription queue and publications continue to be stored on it with no subscriber programs being active. The queue manager deletes the queue (and clears any unretrieved publications from it) only after the application or administrator has chosen to delete the subscription. The subscription can be deleted using an administrative command, or by closing the subscription with the MQCO_REMOVE_SUB option.
   - Consider setting SubExpiry for durable subscriptions so that publications cease to be sent to the queue and the subscriber can consume any remaining publications before removing the subscription and causing the queue manager to delete the queue and any remaining publications on it.
4. Flexible topic string deployment: Subscription topic management is simplified by defining the root part of the subscription using an administratively defined topic. The root part of the topic tree is then hidden from the application. By hiding the root part an application can be deployed without the application inadvertently creating a topic tree that overlaps with another topic tree created by another instance, or another application.
5. Administered topics: by using a topic string in which the first part matches an administratively defined topic object, publications are managed according to the attributes of the topic object.
   - For example, if the first part of the topic string matches the topic string associated with a clustered topic object, then the subscription can receive publications from other members of the cluster
   - The selective matching of administratively defined topic objects and programmatically defined subscriptions enables you to combine the benefits of both. The administrator provides attributes for topics, and the programmer dynamically defines subtopics without being concerned about the management of topics.
   - It is the resultant topic string which is used to match the topic object that provides the attributes associated with the topic, and not necessarily the topic object named in sd.Objectname, although they typically turn out to be one and the same. See "Example 2: Publisher to a variable topic" on page 784.

By making the subscription durable in the example, publications continue to be sent to the subscription queue after the subscriber has closed the subscription with the MQCO_KEEP_SUB option . The queue continues to receive publications when the subscriber is not active. You can override this behavior by creating the subscription with the MQSO_PUBLICATIONS_ON_REQUEST option and using MQSUBRQ to request the retained publication.

The subscription can be resumed later by opening the subscription with the MQCO_RESUME option.

You can use the queue handle, Hobj, returned by MQSUB in a number of ways. The queue handle is used in the example to inquire on the name of the subscription queue. Managed queues are opened using the default model queues SYSTEM.NDURABLE.MODEL.QUEUE or SYSTEM.DURABLE.MODEL.QUEUE. You can override the defaults by providing your own durable and non-durable model queues on a topic by topic basis as properties of the topic object associated with the subscription.

Regardless of the attributes inherited from the model queues, you cannot reuse a managed queue handle to create an additional subscription. Nor can you obtain another handle for the managed queue by opening the managed queue a second time using the returned queue name. The queue behaves as if it has been opened for exclusive input .

Unmanaged queues are more flexible than managed queues. You can, for example share unmanaged queues, or define multiple subscriptions on the one queue. The next example, , demonstrates how to combine subscriptions with an unmanaged subscription queue.

Note: The compact coding style is intended for readability not production use.

The results are shown in Figure 73 on page 793.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

void inquireQname(MQHCONN HConn, MQHOBJ Hobj, MQCHAR48 qName);

int main(int argc, char **argv)
{
    MQCHAR48 topicNameDefault     = "STOCKS";
    char     topicStringDefault[] = "IBM/PRICE";
    MQCHAR48 qmName = "";                 /* Use default queue manager     */
    MQCHAR48 qName = "";                  /* Allocate to query queue name  */
    char     publicationBuffer[101];      /* Allocate to receive messages  */
    char     resTopicStrBuffer[151];   /* Allocate to resolve topic string */

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle             */
    MQHOBJ   Hobj = MQHO_NONE;            /* publication queue handle      */
    MQHOBJ   Hsub = MQSO_NONE;            /* subscription handle           */
    MQLONG   CompCode = MQCC_OK;          /* completion code               */
    MQLONG   Reason = MQRC_NONE;          /* reason code                   */
    MQLONG   messlen = 0;
    MQSD     sd = {MQSD_DEFAULT};         /* Subscription Descriptor       */
    MQMD     md = {MQMD_DEFAULT};         /* Message Descriptor            */
    MQGMO   gmo = {MQGMO_DEFAULT};        /* get message options           */

    char *   topicName   = topicNameDefault;
    char *   topicString = topicStringDefault;
    char *   publication = publicationBuffer;
    char *   resTopicStr = resTopicStrBuffer;
    memset(resTopicStr, 0, sizeof(resTopicStrBuffer));

    switch(argc){               /* Replace defaults with args if provided */
        default:
          topicString = argv[2];
        case(2):
          if (strcmp(argv[1],"/"))       /* "/" invalid = No topic object  */
             topicName = argv[1];
          else
             *topicName = '\0';
        case(1):
             printf("Optional parameters: topicName, topicString\nValues \"%s\"  \"%s\"\n",
                   topicName, topicString);
}
```

Figure 71. Managed MQ subscriber - part 1: declarations and parameter handling.

There are some additional comments to make about the declarations in this example.

MQHOBJ Hobj = MQHO_NONE;
You cannot explicitly open a non-durable managed subscription queue to receive publications, but you do need to allocate storage for the object handle the queue manager returns when it opens the queue for you. It is important to initialize the handle to MQHO_OBJECT. This indicates to the queue manager that it needs to return a queue handle to the subscription queue.

MQSD sd = {MQSD_DEFAULT};
The new subscription descriptor, used in MQSUB.

MQCHAR48 qName;
Although the example doesn't require knowledge of the subscription queue, the example does inquire the name of the subscription queue - the MQINQ binding is a little awkward in the C language, so you might find this part of the example useful to study.

```
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        strncpy(sd.ObjectName, topicName, MQ_TOPIC_NAME_LENGTH);
        sd.ObjectString.VSPtr = topicString;
        sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
        sd.Options = MQSO_CREATE | MQSO_MANAGED | MQSO_NON_DURABLE | MQSO_FAIL_IF_QUIESCING ;
        sd.ResObjectString.VSPtr = resTopicStr;
        sd.ResObjectString.VSBufSize = sizeof(resTopicStrBuffer)-1;
        MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        gmo.Options = MQGMO_WAIT | MQGMO_NO_SYNCPOINT | MQGMO_CONVERT;
        gmo.WaitInterval = 10000;
        inquireQname(Hconn, Hobj, qName);
        printf("Waiting %d seconds for publications matching \"%s\" from \"%-0.48s\"\n",
               gmo.WaitInterval/1000, resTopicStr, qName);
        do {
            memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
            memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
            md.Encoding       = MQENC_NATIVE;
            md.CodedCharSetId = MQCCSI_Q_MGR;
            memset(publicationBuffer, 0, sizeof(publicationBuffer));
            MQGET(Hconn, Hobj, &md, &gmo, sizeof(publicationBuffer)-1),
                publication, &messlen, &CompCode, &Reason);
            if (Reason == MQRC_NONE)
                printf("Received publication \"%s\"\n", publication);
        }
        while (CompCode == MQCC_OK);
        if (CompCode != MQCC_OK && Reason != MQRC_NO_MSG_AVAILABLE) break;
        MQCLOSE(Hconn, &Hsub, MQCO_REMOVE_SUB, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
    return;
}
void inquireQname(MQHCONN Hconn, MQHOBJ Hobj, MQCHAR48 qName) {
#define _selectors 1
#define _intAttrs 1

    MQLONG select[_selectors] = {MQCA_Q_NAME}; /* Array of attribute selectors */
    MQLONG intAttrs[_intAttrs];                /* Array of integer attributes  */
    MQLONG CompCode, Reason;
    MQINQ(Hconn, Hobj, _selectors, select, _intAttrs, intAttrs, MQ_Q_NAME_LENGTH, qName,
          &CompCode, &Reason);
    if (CompCode != MQCC_OK) {
        printf("MQINQ failed with Condition code %d and Reason %d\n", CompCode, Reason);
        strcpy(qName, "unknown queue");
    }
    return;
}
```

Figure 72. Managed MQ subscriber - part 2: code body.

```
W:\Subscribe2\Debug>solution2
Optional parameters: topicName, topicString
Values "STOCKS"  "IBM/PRICE"
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from
"SYSTEM.MANAGED.NDURABLE.48A0AC7403300020    "
Received publication "150"
Completion code 0 and Return code 0

W:\Subscribe2\Debug>solution2 / NYSE/IBM/PRICE
Optional parameters: topicName, topicString
Values ""  "NYSE/IBM/PRICE"
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from
"SYSTEM.MANAGED.NDURABLE.48A0AC7403310020    "
Received publication "150"
Completion code 0 and Return code 0
```

Figure 73. MQ subscriber

There are some additional comments to make about the code in this example.

strncpy(sd.ObjectName, topicName, MQ_Q_NAME_LENGTH);
If topicName is null or blank *(default value)*, the topic name is not used to compute the resolved topic string.

sd.ObjectString.VSPtr = topicString;
Rather than solely use a predefined topic object, in this example the programmer provides a topic object and a topic string, that are combined by MQSUB. Notice the topic string is a MQCHARV structure.

sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
An alternative to setting the length of a MQCHARV field.

sd.Options = MQSO_CREATE | MQSO_MANAGED | MQSO_NON_DURABLE | MQSO_FAIL_IF_QUIESCING;
After defining the topic string, the sd.Options flags need the most careful attention. There are many options, the example specifies only the most commonly used ones. The other options use the default values.

1. As the subscription is *non-durable*, that is, it has a lifetime of the open subscription in the application, set the MQSO_CREATE flag . You can also set the *(default)* MQSO_NON_DURABLE flag for readability.
2. Complementing MQSO_CREATE is MQSO_RESUME. Both flags can be set together; the queue manager either creates a new subscription or resumes an existing subscription, whichever is appropriate. However, if you do specify MQSO_RESUME you must also initialize the MQCHARV structure for sd.SubName, even if there is no subscription to resume. Failure to initialize SubName results in a return code of 2440: MQRC_SUB_NAME_ERROR from MQSUB.

   Note: MQSO_RESUME is always ignored for a non-durable managed subscription: but specifying it without initializing the MQCHARV structure for sd.SubName does cause the error.
3. In addition there is a third flag affecting how the subscription is opened, MQSO_ALTER. Given the correct permissions, the properties of a resumed subscription are changed to match other attributes specified in MQSUB.

   Note: At least one of the MQSO_CREATE, MQSO_RESUME and MQSO_ALTER flags must be specified. See Options (MQLONG). There are examples of using all three flags in "Example 3: Unmanaged MQ subscriber" on page 795.
4. Set MQSO_MANAGED for the queue manager to manage the subscription for you automatically.

sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
Optionally, omit setting the length of MQCHARV for null terminated strings and use the null terminator flag instead.

sd.ResObjectString.VSPtr = resTopicStr;
The resulting topic string is echoed in first printf in the program. Set up MQCHARV ResObjectString for IBM MQ to return the resolved string back to the program.

Note: resTopicStringBuffer is initialized to nulls in memset(resTopicStr, 0, sizeof(resTopicStrBuffer)). Returned topic strings do not end with a trailing null.

sd.ResObjectString.VSBufSize = sizeof(resTopicStrBuffer)-1;
Set the buffer size of the sd.ResObjectString to one less than its actual size. This prevents overwriting the null terminator that is provided, in case the resolved topic string fills the entire buffer.

Note: No error is returned if the topic string is longer than sizeof(resTopicStrBuffer)-1. Even if VSLength > VSBufSiz the length returned in sd.ResObjectString.VSLength is the length of the complete string and not necessarily the length of the returned string. Test sd.ResObjectString.VSLength < sd.ResObjectString.VSBufSiz to confirm the topic string is complete.

MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
The MQSUB function creates a subscription. If it is non-durable you are probably not interested in its name, though you can inspect its status in IBM MQ Explorer. You can provide the sd.SubName parameter as input, so you know what name to look for; you obviously have to avoid name clashes with other subscriptions.

MQCLOSE(Hconn, &Hsub, MQCO_REMOVE_SUB, &CompCode, &Reason);

Closing both the subscription and the subscription queue is optional. In the example the subscription is closed, but not the queue. The MQCLOSE MQCO_REMOVE_SUB option is the default in this case anyway as the subscription is non-durable. Using MQCO_KEEP_SUB is an error.

Note: the subscription *queue* is not closed by MQSUB, and its handle, Hobj, remains valid until the queue is closed by MQCLOSE or MQDISC. If the application terminates prematurely, the queue and subscription are cleaned up by the queue manager sometime after application termination.

Related concepts

"Example 1: MQ Publication consumer" on page 788
The MQ Publication consumer is an IBM MQ message consumer that does not subscribe to topics itself.

"Example 3: Unmanaged MQ subscriber" on page 795
The unmanaged subscriber is an important class of subscriber application. With it, you combine the benefits of publish/subscribe with *control* of queuing and consumption of publications. An unmanaged subscription is where the application is responsible. for specifying the queue where the subscriptions are stored. The example demonstrates different ways of combining subscriptions and queues.

"Writing publisher applications" on page 781
Get started with writing publisher applications by studying two examples. The first is modeled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Example 3: Unmanaged MQ subscriber

The unmanaged subscriber is an important class of subscriber application. With it, you combine the benefits of publish/subscribe with *control* of queuing and consumption of publications. An unmanaged subscription is where the application is responsible. for specifying the queue where the subscriptions are stored. The example demonstrates different ways of combining subscriptions and queues.

The unmanaged pattern is more commonly associated with *durable* subscriptions than *non-durable*. Typically the lifecycle of a subscription created by an unmanaged subscriber is independent of the lifecycle of the subscribing application itself. By making the subscription durable the subscription receives publications even when no subscribing application is active.

You can create durable *managed* subscriptions to achieve the same result, but some applications require more flexibility and control over queues and messages than is possible with a managed subscription. For a durable managed subscription, the queue manager creates a permanent queue for the publications that match the subscription topic. It deletes the queue and associated publications when the subscription is deleted.

Typically durable *managed* subscriptions are used if the lifecycle of the application and the subscription is essentially the same, but hard to guarantee. By making the subscription durable, and having the publisher create persistent publications, there are no lost messages should the queue manager or subscriber terminate prematurely and need to be recovered.

For non-JMS applications, or JMS applications that are not using a shared subscription, the queue manager will implicitly open the durable managed subscription queue for a subscriber in such a way that shared processing of the queue is not possible. In addition, unless your application is using JMS shared subscriptions, it is not possible to create more than one subscription for each managed queue and you might find the queues harder to manage because you have less control over the names of the queues. For these reasons, consider whether the *unmanaged* MQ subscriber is a better fit for applications requiring durable subscriptions than the *managed* MQ subscriber.

The code in Figure 76 on page 800 demonstrates an unmanaged durable subscription pattern. For illustration the code also creates unmanaged, non-durable subscriptions. This example illustrates the following pattern facets:

- On demand subscriptions: the subscription topic strings are dynamic. They are provided by the application when it runs.
- Simplified subscription topic management: subscription topic management is simplified by defining the root part of the subscription topic string using an administratively defined topic. This hides the root part

of the topic tree from the application. By hiding the root part a subscriber can be deployed to different topic trees.

- Flexible subscription management: you can define a subscription either administratively, or create it on-demand in a subscriber program. There is no difference between administratively and programmatically created subscriptions, except an attribute that shows how the subscription was created. There is a third type of subscription that is created automatically by the queue manager for distribution of subscriptions. All subscriptions are displayed in the IBM MQ Explorer.
- Flexible association of subscriptions with queues: a predefined local queue is associated with a subscription by the MQSUB function. There are different ways to use MQSUB to associate subscriptions with queues:
  - Associate a subscription with a queue having *no* existing subscriptions, `MQSO_CREATE + (Hobj from MQOPEN)`.
  - Associate a *new* subscription with a queue having existing subscriptions, `MQSO_CREATE + (Hobj from MQOPEN)`.
  - Move a existing subscription to a different queue, `MQSO_ALTER + (Hobj from MQOPEN)`.
  - Resume an existing subscription associated with an existing queue, `MQSO_RESUME + (Hobj = MQHO_NONE)`, or `MQSO_RESUME + (Hobj = from MQOPEN of queue with existing subscription)`.
  - By combining `MQSO_CREATE | MQSO_RESUME | MQSO_ALTER` in different combinations, you can cater for different input states of the subscription and the queue without having to code multiple versions of MQSUB with different `sd.Options` values.
  - Alternatively, by coding a specific choice of `MQSO_CREATE | MQSO_RESUME | MQSO_ALTER` the queue manager returns an error (Table 123 on page 797) if the states of the subscription and queue provided as input to MQSUB are inconsistent with the value of `sd.Options`. Figure 82 on page 803 shows the results of issuing MQSUB for Subscription X with different individual settings of the `sd.Options` flag, and passing it three different object handles.

  Explore different inputs to the example program in Figure 75 on page 799 to become familiar with these different kinds of errors. One common error, `RC = 2440`, that is not included in the cases listed in the table, is a subscription name error. it is commonly caused by passing a null or invalid subscription name with `MQSO_RESUME` or `MQSO_ALTER`.
- Multiprocessing: You can share among many consumers the work of reading publications. The publications all go onto the single queue associated with the subscription topic. Consumers have a choice of opening the queue directly using MQOPEN or resuming the subscription using MQSUB.
- Subscription concentration: multiple subscriptions can be created on the same queue. Be cautious with this capability as it can lead to overlapping subscriptions, and receiving the same publication multiple times. The `MQSO_GROUP_SUB` option eliminates duplicate publications caused by overlapping subscriptions.
- Subscriber and consumer separation: As well as the three consumer models illustrated in the examples, another model is to separate the consumer from the subscriber. It is a variation of the unmanaged MQ Subscriber, but rather than issue the MQOPEN and MQSUB in the same program, one program subscribes to publications, and another program consumes them. For example, the subscriber might be part of a publish/subscribe cluster and the consumer attached to a queue manager outside the queue manager cluster. The consumer receives publications through standard distributed queuing by defining the subscription queue as a remote queue definition.

Understanding the behavior of `MQSO_CREATE | MQSO_RESUME | MQSO_ALTER` is important, especially if you plan to simplify your code by using combinations of these options. Study the table Table 123 on page 797 that shows the results of passing different queue handles to MQSUB, and the results of running the example program shown in Figure 77 on page 801 to Figure 82 on page 803.

The scenario used to construct the table has one subscription X and two queues, A and B. The subscription name parameter `sd.SubName` is set to X, the name of a subscription attached to queue A. Queue B has no subscription attached to it.

In Table 123 on page 797, MQSUB is passed subscription X and the queue handle to queue A. The results from subscription options are as follows:

- MQSO_CREATE fails because the queue handle corresponds to the queue A which already has a subscription to X. Contrast this behavior to the successful call. That call succeeds because queue B does not have a subscription to X attached to it.
- MQSO_RESUME succeeds because the queue handle corresponds to the queue A which already has a subscription to X. In contrast, the call fails where the subscription X does not exist on queue A.
- MQSO_ALTER behaves in a similar way to MQSO_RESUME with respect to opening the subscription and queue. However if the attributes contained within the subscription descriptor passed to MQSUB differ from the attributes of the subscription, MQSO_RESUME fails, whereas MQSO_ALTER succeeds as long as the program instance has permission to alter the attributes. Note that you can never change the topic string in a subscription; but rather than return an error, MQSUB ignores the topic name and topic string values in the subscription descriptor and uses the values in the existing subscription.

Next, look at Table 123 on page 797 where MQSUB is passed subscription X and the queue handle to queue B. The results from subscription options are as follows:

- MQSO_CREATE succeeds and creates subscription X on queue B because this is a new subscription on queue B.
- MQSO_RESUME fails. MQSUB looks for subscription X on queue B and does not find it, but rather than returning *RC = 2428 - subscription X does not exist*, it returns *RC = 2019 - Subscription queue does not match queue object handle*. The behavior of the third option MQSO_ALTER suggests the reason for this unexpected error. MQSUB expects the queue handle to point to a queue with a subscription. It checks this first before checking whether the subscription named in sd.SubName exists.
- MQSO_ALTER succeeds, and moves the subscription from queue A to queue B.

A case that is not shown in the table is if the subscription name of the subscription on queue A does not match the subscription name in sd.SubName. That call fails with a *RC = 2428 - subscription X does not exist on Queue A*.

Table 123. Errors from MQSUB with different queue handles and subscription combinations

Queue handles	**Queue A** **Subscription X** **Queue B** **No subscription**	**Queue A** **No subscription** **Queue B** **No subscription**
Hobj for *Queue A* passed to MQSUB	**MQSO_CREATE** RC = 2432 - Subscription X already exists on Queue A **MQSO_RESUME** Resumes subscription X on Queue A **MQSO_ALTER** Resumes subscription X on Queue A and makes permitted alterations	**MQSO_CREATE** Creates subscription X on Queue A **MQSO_RESUME** RC = 2428 - Subscription X does not exist on Queue A **MQSO_ALTER** RC = 2428 - Subscription X does not exist on Queue A
Hobj for *Queue B* passed to MQSUB	**MQSO_CREATE** Creates new subscription X on Queue B **MQSO_RESUME** RC = 2019 - Subscription queue does not match queue object handle **MQSO_ALTER** Move subscription X from Queue A to Queue B	**MQSO_CREATE** Creates new subscription X on Queue B **MQSO_RESUME** RC = 2428 - subscription X does not exist on Queue B **MQSO_ALTER** RC = 2428 - subscription X does not exist on Queue B

Table 123. Errors from MQSUB with different queue handles and subscription combinations (continued)		
Queue handles	**Queue A** **Subscription X** **Queue B** **No subscription**	**Queue A** **No subscription** **Queue B** **No subscription**
MQHO_NONE passed to MQSUB	**MQSO_CREATE** RC = 2019 - Bad object handle: set MQSO_MANAGED flag to create a managed subscription and create a managed queue **MQSO_RESUME** Resumes subscription X on Queue A and returns Hobj to Queue A **MQSO_ALTER** Resumes subscription X on Queue A, returns Hobj to Queue A and makes permitted alterations	**MQSO_CREATE** RC = 2019 - Bad object handle: set MQSO_MANAGED flag to create a managed subscription and create a managed queue **MQSO_RESUME** RC = 2428 - No subscription X **MQSO_ALTER** RC = 2019 - Bad object handle: No queue A or B

Note: The compact coding style is intended for readability not production use.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

void inquireQname(MQHCONN HConn, MQHOBJ Hobj, MQCHAR48 qName);

int main(int argc, char **argv)
{
    MQCHAR48 topicNameDefault            = "STOCKS";
    char     topicStringDefault[]        = "IBM/PRICE";
    char     subscriptionNameDefault[]   = "IBMSTOCKPRICESUB";
    char     subscriptionQueueDefault[]  = "STOCKTICKER";
    char     publicationBuffer[101];      /* Allocate to receive messages     */
    char     resTopicStrBuffer[151];      /* Allocate to resolve topic string */
    MQCHAR48 qmName = "";                 /* Default queue manager            */
    MQCHAR48 qName = "";                  /* Allocate storage for MQINQ       */

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle                */
    MQHOBJ   Hobj = MQHO_NONE;            /* subscription queue handle        */
    MQHOBJ   Hsub = MQSO_NONE;            /* subscription handle              */
    MQLONG   CompCode = MQCC_OK;          /* completion code                  */
    MQLONG   Reason = MQRC_NONE;          /* reason code                      */
    MQLONG   messlen = 0;
    MQOD     od = {MQOD_DEFAULT};         /* Unmanaged subscription queue     */
    MQSD     sd = {MQSD_DEFAULT};         /* Subscription Descriptor          */
    MQMD     md = {MQMD_DEFAULT};         /* Message Descriptor               */
    MQGMO   gmo = {MQGMO_DEFAULT};        /* get message options              */
    MQLONG   sdOptions = MQSO_CREATE | MQSO_RESUME | MQSO_DURABLE |
MQSO_FAIL_IF_QUIESCING;

    char *   topicName = topicNameDefault;
    char *   topicString = topicStringDefault;
    char *   subscriptionName =  subscriptionNameDefault;
    char *   subscriptionQueue = subscriptionQueueDefault;
    char *   publication = publicationBuffer;
    char *   resTopicStr = resTopicStrBuffer;
    memset(resTopicStrBuffer, 0, sizeof(resTopicStrBuffer));
```

Figure 74. Unmanaged MQ subscriber - part 1: declarations.

```
    switch(argc){                   /* Replace defaults with args if provided */
default:
        switch((argv[5][0])) {
case('A'):  sdOptions =  MQSO_ALTER  | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;
            break;
case('C'):  sdOptions =  MQSO_CREATE | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;
            break;
case('R'):  sdOptions =  MQSO_RESUME | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;
            break;
default:    ;
        }
case(5):
    if (strcmp(argv[4],"/")) /* "/" invalid = No subscription  */
        subscriptionQueue = argv[4];
    else {
        *subscriptionQueue = '\0';
        if (argc > 5) {
            if (argv[5][0] == 'C') {
                sdOptions = sdOptions + MQSO_MANAGED;
            }
        }
        else
            sdOptions = sdOptions + MQSO_MANAGED;
    }

case(4):
    if (strcmp(argv[3],"/")) /* "/" invalid = No subscription  */
        subscriptionName = argv[3];
    else {
        *subscriptionName = '\0';
        sdOptions = sdOptions - MQSO_DURABLE;
    }
case(3):
    if (strcmp(argv[2],"/")) /* "/" invalid = No topic string  */
        topicString = argv[2];
    else
        *topicString = '\0';
case(2):
    if (strcmp(argv[1],"/")) /* "/" invalid = No topic object  */
        topicName = argv[1];
    else
        *topicName = '\0';
case(1):
    sd.Options = sdOptions;
    printf("Optional parameters: "
    printf("topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|
R(esume)\n");
    printf("Values \"%-.48s\"  \"%s\" \"%s\"  \"%-.48s\" sd.Options=%d\n",
           topicName, topicString, subscriptionName, subscriptionQueue, sd.Options);
}
```

Figure 75. Unmanaged MQ subscriber - part 2: parameter handling.

Additional comments about the parameter handling in this example are as follows:

switch((argv[5][0]))
You have the choice of entering A lter | C reate | R esume in parameter 5, to test the effect of overriding part of the MQSUB option setting used by default in the example. The default setting used by the example is MQSO_CREATE | MQSO_RESUME | MQSO_DURABLE.

Note: Setting MQSO_ALTER or MQSO_RESUME without setting MQSO_DURABLE is an error, and sd.SubName must be set and refer to a subscription that can be resumed or altered.

***subscriptionQueue = '\0';**
sdOptions = sdOptions + MQSO_MANAGED;
If the default subscription queue, STOCKTICKER is replaced by a null string then as long as MQSO_CREATE is set, the example sets the MQSO_MANAGED flag and creates a dynamic subscription queue. If Alter or Resume are set in the fifth parameter the behavior of the example will depend on the value of subscriptionName.

***subscriptionName = '\0';**
sdOptions = sdOptions - MQSO_DURABLE;

If the default subscription, IBMSTOCKPRICESUB, is replaced by a null string then the example removes the MQSO_DURABLE flag. If you run the example providing the default values for the other parameters an additional temporary subscription destined to STOCKTICKER is created and receives duplicate publications. Next time you run the example, without any parameters, you receive just one publication again.

```
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        if (strlen(subscriptionQueue)) {
            strncpy(od.ObjectName, subscriptionQueue, MQ_Q_NAME_LENGTH);
            MQOPEN(Hconn, &od, MQOO_INPUT_AS_Q_DEF | MQOO_FAIL_IF_QUIESCING | MQOO_INQUIRE,
                   &Hobj, &CompCode, &Reason);
            if (CompCode != MQCC_OK) break;
        }
        strncpy(sd.ObjectName, topicName, MQ_TOPIC_NAME_LENGTH);
        sd.ObjectString.VSPtr = topicString;
        sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
        sd.SubName.VSPtr = subscriptionName;
        sd.SubName.VSLength = MQVS_NULL_TERMINATED;
        sd.ResObjectString.VSPtr = resTopicStr;
        sd.ResObjectString.VSBufSize = sizeof(resTopicStrBuffer)-1;
        MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        gmo.Options = MQGMO_WAIT | MQGMO_NO_SYNCPOINT | MQGMO_CONVERT;
        gmo.WaitInterval = 10000;
        gmo.MatchOptions = MQMO_MATCH_CORREL_ID;
        memcpy(md.CorrelId, sd.SubCorrelId, MQ_CORREL_ID_LENGTH);
        inquireQname(Hconn, Hobj, qName);
        printf("Waiting %d seconds for publications matching \"%s\" from %-0.48s\n",
               gmo.WaitInterval/1000, resTopicStr, qName);
        do {
            memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
            memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
            md.Encoding       = MQENC_NATIVE;
            md.CodedCharSetId = MQCCSI_Q_MGR;
            MQGET(Hconn, Hobj, &md, &gmo, sizeof(publication), publication, &messlen,
&CompCode, &Reason);
            if (Reason == MQRC_NONE)
                printf("Received publication \"%s\"\n", publication);
        }
        while (CompCode == MQCC_OK);
        if (CompCode != MQCC_OK && Reason != MQRC_NO_MSG_AVAILABLE) break;
        MQCLOSE(Hconn, &Hsub, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
void inquireQname(MQHCONN Hconn, MQHOBJ Hobj, MQCHAR48 qName) {
#define _selectors 1
#define _intAttrs 1

    MQLONG select[_selectors] = {MQCA_Q_NAME}; /* Array of attribute selectors    */
    MQLONG intAttrs[_intAttrs];            /* Array of integer attributes            */
    MQLONG CompCode, Reason;
    MQINQ(Hconn, Hobj, _selectors, select, _intAttrs, intAttrs, MQ_Q_NAME_LENGTH, qName,
&CompCode, &Reason);
    if (CompCode != MQCC_OK) {
        printf("MQINQ failed with Condition code %d and Reason %d\n", CompCode, Reason);
        strncpy(qName, "unknown queue", MQ_Q_NAME_LENGTH);
    }
    return;
}
```

Figure 76. Unmanaged MQ subscriber - part 3: code body.

Additional comments on the code in this example are as follows:

if (strlen(subscriptionQueue))
If there is no subscription queue name then the example uses MQHO_NONE as the value of Hobj.

MQOPEN(...);
The subscription queue is opened and the queue handle saved in Hobj.

MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
The subscription is opened using the Hobj passed from MQOPEN (or MQHO_NONE if there is no subscription queue name). An unmanaged queue can be resumed without explicitly opening it with an MQOPEN.

MQCLOSE(Hconn, &Hsub, MQCO_NONE, &CompCode, &Reason);
The subscription is closed using the subscription handle. Depending on whether the subscription is durable or not, the subscription is closed with an implicit MQCO_KEEP_SUB or MQCO_REMOVE_SUB. You can close a durable subscription with MQCO_REMOVE_SUB, but you cannot close a non-durable subscription with MQCO_KEEP_SUB. The action of MQCO_REMOVE_SUB is to remove the subscription which stops any further publications being sent to the subscription queue.

MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
No special action is taken if the subscription is unmanaged. If the queue is managed and the subscription closed with either an explicit or implicit MQCO_REMOVE_SUB, then all publications are purged from the queue and queue deleted at this point.

gmo.MatchOptions = MQMO_MATCH_CORREL_ID;
memcpy(md.CorrelId, sd.SubCorrelId, MQ_CORREL_ID_LENGTH);
Ensure that the messages received are those for our subscription.

Results from the example illustrate aspects of publish/subscribe:

In Figure 77 on page 801 the example starts by publishing 130 on the NYSE/IBM/PRICE topic.

```
W:\Subscribe3\Debug>..\..\Publish2\Debug\publishstock
Provide parameters: TopicObject TopicString Publication
Publish "130" to topic "STOCKS" and topic string "IBM/PRICE"
Published "130" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0
```

Figure 77. Publish 130 to NYSE/IBM/PRICE

In Figure 78 on page 801 execution of the example using default parameters receives the retained publication 130. The provided topic object and topic string are ignored, as shown in Figure 82 on page 803. The topic object and topic string are always taken from the subscription object, when one is provided, and the topic string is immutable. The actual behavior of the example depends on the choice or combination of MQSO_CREATE, MQSO_RESUME, and MQSO_ALTER. In this example MQSO_RESUME is the option selected.

```
W:\Subscribe3\Debug>solution3
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|
C(reate)|R(esume)
Values "STOCKS" "IBM/PRICE" "IBMSTOCKPRICESUB" "STOCKTICKER" sd.Options=8206
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Completion code 0 and Return code 0
```

Figure 78. Receive the retained publication

In (Figure 79 on page 802) no publications are received, because the durable subscription has already received the retained publication. In this example, the subscription is resumed by providing only the subscription name without the queue name. If the queue name was provided, the queue would be opened first and the handle passed to MQSUB.

Note: The 2038 error from MQINQ is due to the implicit MQOPEN of STOCKTICKER by MQSUB not including the MQOO_INQUIRE option. Avoid the 2038 return code from MQINQ by opening the queue explicitly.

```
W:\Subscribe3\Debug>solution3 STOCKS IBM/PRICE IBMSTOCKPRICESUB / Resume
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|
C(reate)|R(esume)
Values "STOCKS" "IBM/PRICE" "IBMSTOCKPRICESUB" "" sd.Options=8204
MQINQ failed with Condition code 2 and Reason 2038
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from unknown queue
Completion code 0 and Return code 0
```

Figure 79. Resume subscription

In Figure 80 on page 802, the example creates a non-durable unmanaged subscription using STOCKTICKER as the destination. Because this is a new subscription, it receives the retained publication.

```
W:\Subscribe3\Debug>solution3 STOCKS IBM/PRICE / STOCKTICKER Create
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|
C(reate)|R(esume)
Values "STOCKS" "IBM/PRICE" "" "STOCKTICKER" sd.Options=8194
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Completion code 0 and Return code 0
```

Figure 80. Receive retained publication with new unmanaged non durable subscription

In Figure 81 on page 802, to demonstrate overlapping subscriptions, another publication is sent, changing the retained publication. Next, a new non-durable, unmanaged subscription is created by not providing a subscription name. The retained publication is received twice, once for the new subscription, and once for the durable IBMSTOCKPRICESUB subscription that is still active on the STOCKTICKER queue. The example is an illustration that it is the queue that has subscriptions, and not the application. Despite not referring to the IBMSTOCKPRICESUB subscription in this invocation of the application, the application receives the publication twice: once from the durable subscription that was created administratively, and once from the non-durable subscription created by the application itself.

```
W:\Subscribe3\Debug>..\..\Publish2\Debug\publishstock
Provide parameters: TopicObject TopicString Publication
Publish "130" to topic "STOCKS" and topic string "IBM/PRICE"
Published "130" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0

W:\Subscribe3\Debug>solution3 STOCKS IBM/PRICE / STOCKTICKER Create
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|
C(reate)|R(esume)
Values "STOCKS" "IBM/PRICE" "" "STOCKTICKER" sd.Options=8194
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Received publication "130"
Completion code 0 and Return code 0
```

Figure 81. Overlapping subscriptions

In Figure 82 on page 803 the example demonstrates that providing a new topic string and an existing subscription does not result in a changed subscription.

1. In the first case, `Resume` resumes the existing subscription, as you might expect, and ignores the changed topic string.
2. In the second case, `Alter` causes an error, `RC = 2510, Topic not alterable`.
3. In the third example, `Create` causes an error `RC = 2432, Sub already exists`.

```
W:\Subscribe3\Debug>solution3 "" NASDAC/IBM/PRICE IBMSTOCKPRICESUB STOCKTICKER Resume
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "" "NASDAC/IBM/PRICE" "IBMSTOCKPRICESUB" "STOCKTICKER" sd.Options=8204
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Completion code 0 and Return code 0

W:\Subscribe3\Debug>solution3 "" NASDAC/IBM/PRICE IBMSTOCKPRICESUB STOCKTICKER Alter
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "" "NASDAC/IBM/PRICE" "IBMSTOCKPRICESUB" "STOCKTICKER" sd.Options=8201
Completion code 2 and Return code 2510

W:\Subscribe3\Debug>solution3 "" NASDAC/IBM/PRICE IBMSTOCKPRICESUB STOCKTICKER Create
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "" "NASDAC/IBM/PRICE" "IBMSTOCKPRICESUB" "STOCKTICKER" sd.Options=8202
Completion code 2 and Return code 2432
```

Figure 82. Subscription topics cannot be changed

Related concepts

“Example 1: MQ Publication consumer” on page 788
The MQ Publication consumer is an IBM MQ message consumer that does not subscribe to topics itself.

“Example 2: Managed MQ subscriber” on page 790
The managed MQ subscriber is the preferred pattern for most subscriber applications. A managed subscription is one where IBM MQ handles the subscription and does the registering and de-registering for you. The example requires *no* administrative definition of queues, topics or subscriptions.

“Writing publisher applications” on page 781
Get started with writing publisher applications by studying two examples. The first is modeled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Publish/subscribe lifecycles

Consider the lifecycles of topics, subscriptions, subscribers, publications, publishers and queues in designing publish/subscribe applications.

The lifecycle of an object, such as a subscription, starts with its creation and ends with its deletion. It may also include other states and changes that it goes through, such as temporary suspension, having parent and children topics, expiration and deletion.

Traditionally IBM MQ objects such as queues are created administratively, or by administrative programs using Programmable Command Format (PCF). Publish/subscribe is different in providing the MQSUB and MQCLOSE API verbs to create and delete subscriptions, having the concept of managed subscriptions that not only create and delete queues, but also clean up unconsumed messages, and having associations between administratively created topic objects and programmatically or administratively created topic strings.

This functional richness caters for a wide range of publish/subscribe requirements, and also simplifies designing some common patterns of publish/subscribe application. Managed subscriptions, for example, simplify both the programming and administration of a subscription that is intended to last only as long as the program that created it. Unmanaged subscriptions simplify programming where there is a looser connection between subscribing and consuming publications. Centrally created subscriptions are useful where the pattern is one of routing publication traffic to consumers based on a centralized model of control, for example sending flight information to automated gates, whereas programmatically created subscriptions might be used if gate staff are responsible for subscribing to the passengers records for that flight, by entering a flight number at a gate.

In this last example a managed durable subscription might be appropriate: managed, because the subscriptions are being created very often, and have a clear endpoint when the gate closes and the subscription can be programmatically removed; durable, to avoid losing a passenger record due to the gate subscriber program going down for one reason or another [8] . To initiate the publication of passenger records to the gate, a possible design would be for the gate application to both subscribe to the passenger records using the gate number, and publish the gate opening event using the gate number. The publisher responds to the gate opening event by publishing the passenger records - which might then also go to

[8] The publisher must send the passenger records as persistent messages to avoid other possible failures, of course.

other interested parties, such as billing, to record the flight is taking place, and to customer services, to text notifications to passengers' mobile phones of the gate number.

The centrally managed subscription might use a durable unmanaged model, routing passenger lists to the gate using a predefined queue for each gate.

The following three examples of publish/subscribe lifecycles illustrate how managed non-durable, managed durable, and unmanaged durable subscribers interact with subscriptions, topics, queues, publishers and the queue manager, and how the responsibilities might be divided between administration and the subscriber programs.

Managed non-durable subscriber

Figure 83 on page 805 shows an application creating a managed non-durable subscription, getting two messages that are published to the topic identified in the subscription, and terminating. The interactions labeled in an italic gray font with dotted arrows are implicit.

There are some points to note.

1. The application creates a subscription on a topic that has already been published to twice. When the subscriber receives its first publication, it receives the *second* publication which is the currently retained publication.
2. The queue manager creates a temporary subscription queue as well as creating a subscription for the topic.
3. The subscription has an expiry. When the subscription expires no more publications on the topic are sent to this subscription, but the subscriber continues to get messages published before the subscription expired. Publication expiry is not affected by subscription expiry.
4. The fourth publication is not placed on the subscription queue and consequently the last `MQGET` does not return a publication.
5. Although the subscriber closes its subscription, it does not close its connection to the queue or the queue manager.
6. The queue manager cleans up shortly after the application terminates. Because the subscription is managed and non-durable, the subscription queue is deleted.

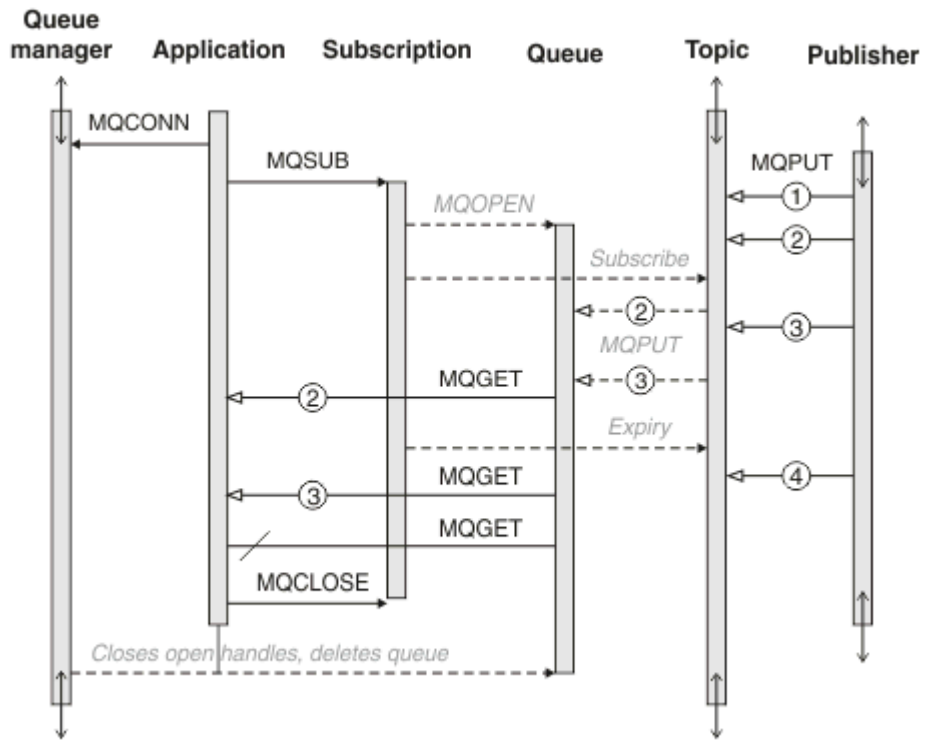


Figure 83. Managed non-durable subscriber lifelines

Managed durable subscriber

The managed durable subscriber takes the previous example a step further, and shows a managed subscription surviving the termination and restart of the subscribing application.

There are some new points to note.

1. In this example, unlike the last, the publication topic did not exist before it was defined in the subscription.
2. The first time the subscriber terminates, it closes the subscription with the option MQCO_KEEP_SUB. That is the default behavior for implicitly closing a managed durable subscription.
3. When the subscriber resumes the subscription, the subscription queue is reopened.
4. The new publication 2, placed on the queue before it is reopened, is available to MQGET, even after the subscription has been removed.

   Even though the subscription is durable, the subscriber reliably receives all messages sent by the publisher only if *both* the subscription is durable and the messages persistent. Message persistence depends on the setting of the Persistent field in the MQMD of the message sent by the publisher. A subscriber has no control over this.
5. Closing the subscription with the flag MQCO_REMOVE_SUB removes the subscription, stopping any further publications being placed on the subscription queue. When the subscription queue is closed, then the queue manager removes the unread publication 3, and then deletes the queue. The action is equivalent to administratively deleting the subscription.

   Note: Do not delete the queue manually, or issue MQCLOSE with the option MQCO_DELETE, or MQCO_PURGE_DELETE. The visible implementation details of a managed subscription is not part of the supported IBM MQ interface. The queue manager manage cannot manage a subscription reliably unless it has complete control.

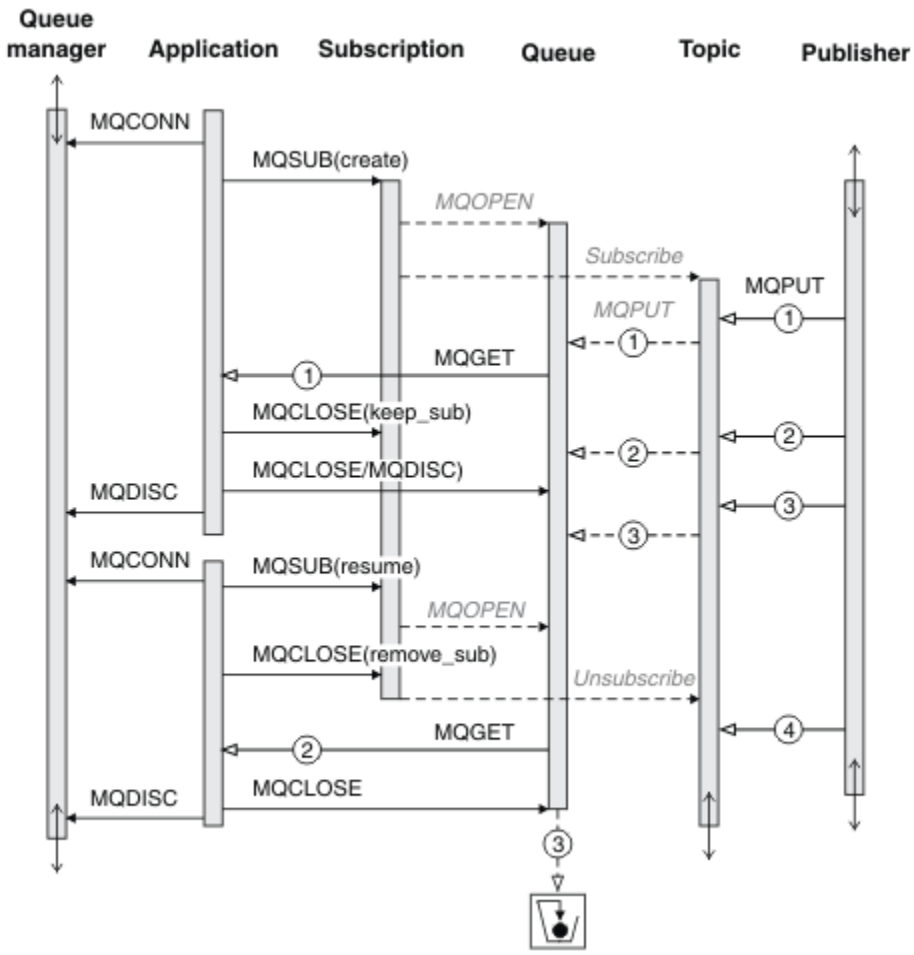


Figure 84. Managed durable subscriber lifelines

Unmanaged durable subscriber

An administrator is added in the third example: the unmanaged durable subscriber. It is a good example to show how the administrator might interact with a publish/subscribe application.

The points to note are listed.

1. The publisher puts a message, 1, to a topic that later becomes associated with the topic object that is used for subscription. The topic object defines a topic string that matches the topic that was published to by using wildcards.
2. The topic has a retained publication.
3. The administrator creates a topic object, a queue and a subscription. The topic object and queue need to be defined before the subscription.
4. The application opens the queue associated with the subscription and passes MQSUB the handle of the queue. It could, alternatively, simply open the subscription, passing it the queue handle MQHO_NONE. The converse is not true, it cannot resume a subscription by passing it only queue handle without a subscription name - a queue might have multiple subscriptions.
5. The application opens the subscription using the option MQSO_RESUME even though it is the first time it has opened the subscription. It is resuming an administratively created subscription.
6. The subscriber receives the retained publication, 1. Publication 2, although published before any publications were received by the subscriber, was published after the subscription started, and is the second publication on the subscription queue.

Note: If the retained publication is not published as a persistent message, then it is lost after queue manager restart.

7. In this example the subscription is durable. It is possible for a program to create an unmanaged non-durable subscription; it should be obvious this is not something an administrator can do.
8. The effect of the option `MQCO_REMOVE_SUB` on closing the subscription is to remove the subscription just as if the administrator had deleted it. This stops any further publications being sent to the queue, but does not affect publications that are already on the queue, even when the queue is closed, unlike a *managed* durable subscription.
9. The administrator later deletes the remaining message, 3, and deletes the queue.

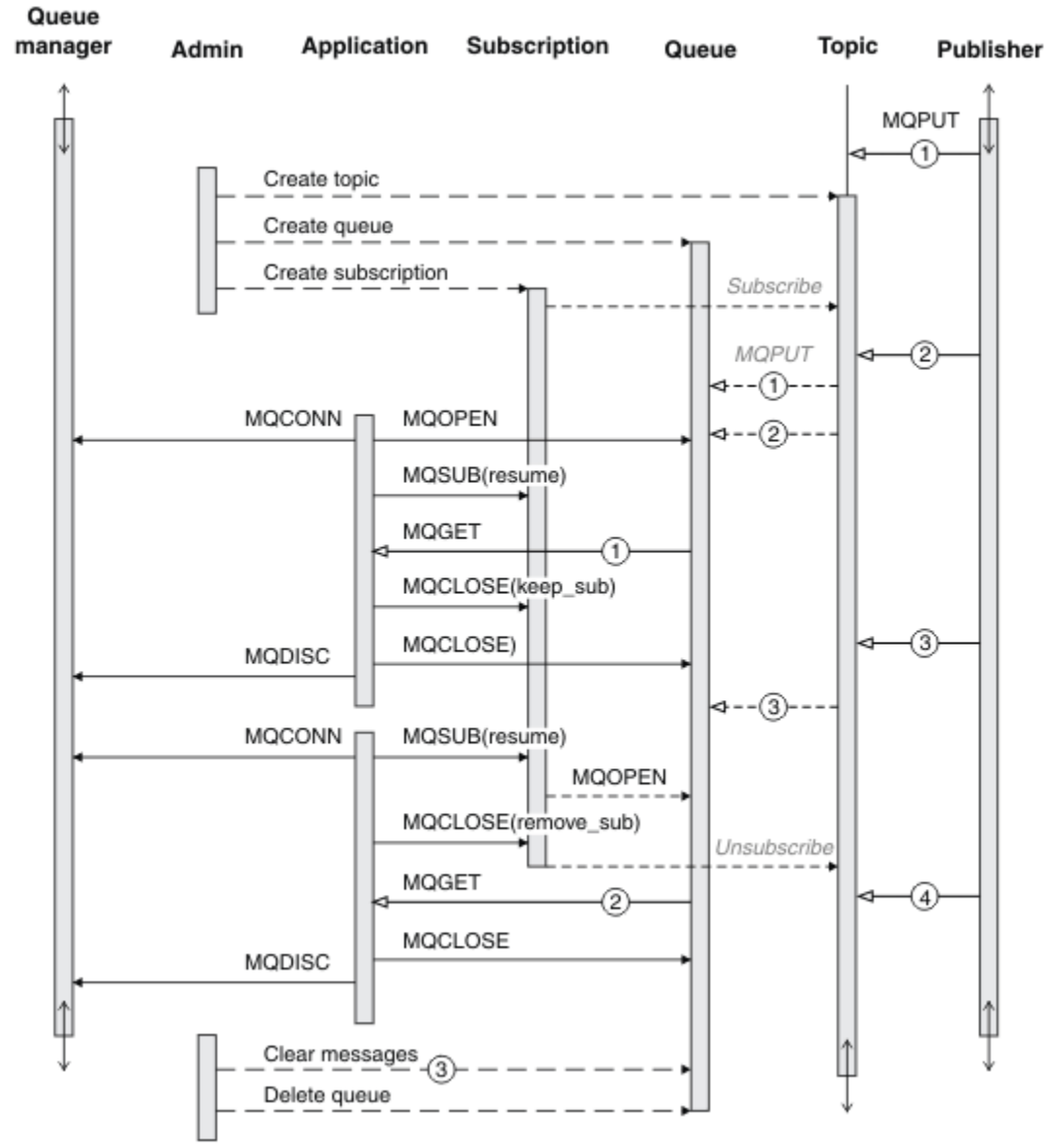


Figure 85. Unmanaged durable subscriber lifelines

A normal pattern for an unmanaged subscription is for queue and subscription housekeeping to be performed by the administrator. Typically one would not attempt to emulate the behavior of a managed subscriber and tidy up queues and subscriptions programmatically in application code. If you find yourself needing to write management logic, question whether you can achieve the same results using a managed pattern. It is not easy to write tightly synchronized, completely reliable management code. It is easier to

tidy up later, either manually, or using a automated management program, when you can be sure that messages, subscriptions, and queues can be simply deleted, regardless of their state.

Publish/subscribe message properties

Several message properties relate to IBM MQ publish/subscribe messaging.

PubAccountingToken

This is the value that will be in the AccountingToken field of the Message Descriptor (MQMD) of all publication messages matching this subscription. AccountingToken is part of the identity context of the message. For more information about message context, see “Message context” on page 45. For more information about the AccountingToken field in the MQMD, see AccountingToken.

PubApplIdentityData

This is the value that will be in the ApplIdentityData field of the Message Descriptor (MQMD) of all publication messages matching this subscription. ApplIdentityData is part of the identity context of the message. For more information about message context, see “Message context” on page 45. For more information about the ApplIdentityData field in the MQMD, see ApplIdentityData.

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

If the option MQSO_SET_IDENTITY_CONTEXT is specified, the PubApplIdentityData is being generated by the user and this field is an input field which contains the ApplIdentityData to be set in each publication for this subscription.

PubPriority

This is the value that will be in the Priority field of the Message Descriptor (MQMD) of all publication messages matching this subscription. For more information about the Priority field in the MQMD, see Priority.

The value must be greater than or equal to zero; zero is the lowest priority. The following special values can also be used:

- MQPRI_PRIORITY_AS_Q_DEF - When a subscription queue is provided in the Hobj field in the MQSUB call, and is not a managed handle, then the priority for the message is taken from the DefProirity attribute of this queue. If the queue so identified is a cluster queue, or there is more than one definition in the queue-name resolution path, the priority is determined when the publication message is put to the queue as described for Priority in the MQMD. If the MQSUB call uses a managed handle, the priority for the message is taken from the DefPriority attribute of the model queue associated with the topic subscribed to.
- MQPRI_PRIORITY_AS_PUBLISHED - The priority for the message is the priority of the original publication. This is the initial value of this field.

SubCorrelId

Attention: a correlation identifier can only be passed between queue managers in a publish/subscribe cluster, not a hierarchy.

All publications sent to match this subscription will contain this correlation identifier in the message descriptor. If multiple subscriptions use the same queue to get their publications from, using MQGET by correlation ID allows only publications for a specific subscription to be obtained. This correlation identifier can either be generated by the queue manager or by the user.

If the option MQSO_SET_CORREL_ID is not specified, the correlation identifier is generated by the queue manager and this field is an output field which contains the correlation identifier which will be set in each message published for this subscription.

If the option MQSO_SET_CORREL_ID is specified, the correlation identifier is being generated by the user and this field is an input field which contains the correlation identifier to be set in each publication for this subscription. In this case, if the field contains MQCI_NONE, the correlation identifier which will be set in each message published for this subscription will be the correlation identifier created by the original put of the message.

If the option MQSO_GROUP_SUB is specified and the correlation identifier specified is the same as an existing grouped subscription using the same queue and an overlapping topic string, only the most significant subscription in the group is provided with a copy of the publication.

SubUserData

This is the subscription user data. The data provided on the subscription in this field will be included as the MQSubUserData message property of every publication sent to this subscription.

Publication properties

Table 124 on page 809 lists the publication properties that are provided with a publication message.

You can access these properties directly from the **MQRFH2** folder, or retrieve them using MQINQMP. MQINQMP accepts either the property name or **MQRFH2** name as the name of the property to inquire on.

Table 124. Publication properties

Property name	MQRFH2 name	Type	Description
MQTopicString	mqps.Top	MQTYPE_STRING	Topic string
MQSubUserData	mqps.Sud	MQTYPE_STRING	Subscriber user data
MQIsRetained	mqps.Ret	MQTYPE_BOOLEAN	Retained publication
MQPubOptions	mqps.Pub	MQTYPE_INT32	Publication options
MQPubLevel	mqps.Pbl	MQTYPE_INT32	Publication level
MQPubTime	mqpse.Pts	MQTYPE_STRING	Publication time
MQPubSeqNum	mqpse.Seq	MQTYPE_INT32	Publication sequence number
MQPubStrIntData	mqpse.Sid	MQTYPE_STRING	String/Integer data added by the publisher
MQPubFormat	mqpse.Pfmt	MQTYPE_INT32	Message format: MQRFH1 MQRFH2 PCF

Message ordering

For a particular topic, messages are published by the queue manager in the same order as they are received from publishing applications (subject to reordering based on message priority).

Message ordering normally means that each subscriber receives messages from a particular queue manager, on a particular topic, from a particular publisher in the order that they are published by that publisher.

However, as with all IBM MQ messages, it is possible for messages, occasionally, to be delivered out of order. This can happen in the following situations:

- If a link in the network goes down and subsequent messages are rerouted along another link
- If a queue becomes temporarily full, or put-inhibited, so that a message is put to a dead-letter queue and therefore delayed, while subsequent messages pass straight through.

- If the administrator deletes a queue manager when publishers and subscribers are still operating, causing queued messages to be put to the dead-letter queue and subscriptions to be interrupted.

If these circumstances cannot occur, publications are always delivered in order.

Note: It is not possible to use grouped or segmented messages with Publish/Subscribe.

Intercepting publications

You can intercept a publication, modify it, and then republish it before it reaches any other subscriber.

You might want to intercept a publication before it reaches a subscriber in order to do one of the following actions:

- Attach additional information to the message
- Block the message
- Transform the message

You can perform the same operation on each message or vary the operation depending on the subscription, the message, or the message header.

Related reference

MQ_PUBLISH_EXIT - Publish exit

Subscription levels

Set the subscription level of a subscription to intercept a publication before it reaches its final subscribers. An intercepting subscriber subscribes at a higher subscription level, and republishes at a lower publication level. Build a chain of intercepting subscribers to perform message processing on a publication before it is delivered to final subscribers.

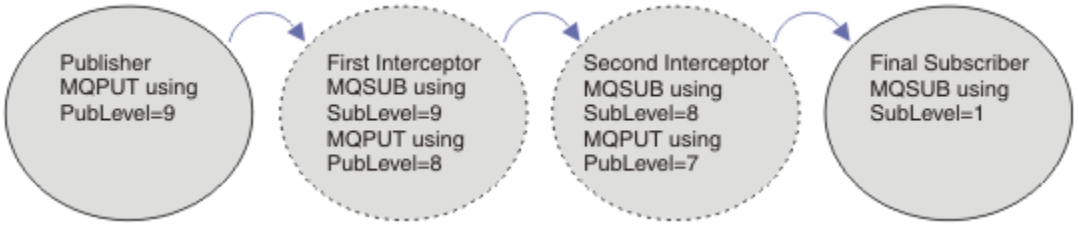


Figure 86. Sequence of intercepting subscribers

To intercept a publication, use the **MQSD** `SubLevel` attribute. After a message has been intercepted, it can be transformed and then republished at a lower publication level by changing the **MQPMO** `PubLevel` attribute. The message then goes to the final subscribers, or it is intercepted again by an intermediate subscriber at a lower subscription level.

The intercepting subscriber typically transforms a message before republishing it. A sequence of intercepting subscribers forms a message flow. Alternatively, you might not republish the intercepted publication: Subscribers at lower subscription levels would not receive the message.

Ensure that the interceptor receives publications before any other subscribers. Set the subscription level of the interceptor higher than other subscribers. By default, subscribers have a `SubLevel` of 1. The highest value is 9. A publication must start with a `PubLevel` at least as high as the highest `SubLevel`. Publish initially with the default `PubLevel` of 9.

- If you have one intercepting subscriber on a topic, set the `SubLevel` to 9.
- For multiple intercepting applications on a topic, set a lower `SubLevel` for each successive intercepting subscriber.
- You can implement a maximum of 8 intercepting applications, with subscription levels from 9 down to 2 inclusive. The final recipient of the message has a `SubLevel` of 1.

The interceptor with the highest subscription level that is equal to, or lower than, the `PubLevel` of the publication receives the publication first. Configure only one intercepting subscriber for a topic at

a particular subscription level. Having multiple subscribers at a particular subscription level results in multiple copies of the publication being sent to the final set of subscribing applications.

A subscriber with a SubLevel of 0 is used as a catchall. It receives the publication if no final subscriber gets the message. A subscriber with SubLevel of 0 might be used to monitor the publications that no other subscribers received.

Programming an intercepting subscriber

Use the subscription options described in Table 125 on page 811.

Table 125. Subscription options for intercepting subscribers

Subscription option	Notes
MQSO_SET_CORREL_ID and SubCorrelId set to MQCI_NONE	Keep the CorrelId of the intercepted publication the same as the original publication. **Note:** You cannot pass the correlation identifier of a publication in a hierarchy. The field is used by the queue manager.
PubPriority set to MQPRI_PRIORITY_AS_PUBLISHED	Keep the priority of the intercepted publication the same as the original publication.

The options in Table 125 on page 811 must be used by all the intercepting subscribers. The result is that the correlation identifier and message priority are not modified from the setting of the original publisher.

When the intercepting subscriber has processed the publication, it republishes the message to the same topic at a PubLevel one lower than the SubLevel of its own subscription. If the intercepting subscriber set a SubLevel of 9, it republishes the message with a PubLevel of 8.

To republish the message correctly, several pieces of information from the original publication are required. Reuse the same **MQMD** as in the original message and set MQPMO_PASS_ALL_CONTEXT to ensure all information in the **MQMD** is passed on to the next subscriber. Copy the values from the message properties shown in Table 126 on page 811 into the corresponding fields of the republished message. The intercepting subscriber can change these values. Use the OR operator to add additional values to the **MQPMO**. Options field, to combine the put message options.

You must open the publication queue explicitly rather than use a managed publication queue. You cannot set MQSO_SET_CORREL_ID for a managed queue. You also cannot set MQOO_SAVE_ALL_CONTEXT on a managed queue. See the code fragments listed in “Examples” on page 812.

Table 126. MQPUT values for republished messages

Republish message using MQPUT	Information in publication message
MQOD. ObjectString	Message property MQTopicString
MQPMO. Options	Message property MQPubOptions

The final subscriber has the choice of setting its subscription options differently. For example, it might set the publication priority explicitly rather than to MQPRI_PRIORITY_AS_PUBLISHED. The settings of a final subscriber only affect publication from the final intercepting subscriber in the chain.

Retained publications

A retained publication must be preserved after it has been intercepted, by copying the original put-message options into the republished message.

The MQPMO_RETAIN option is set by the publisher. Each intercepting subscriber must transfer the MQPubOptions to the put-message options of the republished message as shown in Table 126 on

page 811. Copying the put-message options preserves the options set by the original publisher, including whether to retain the publication.

When a publication finishes its passage down the chain of intercepting subscribers, and is delivered to final subscribers, it is finally retained. New subscribers, at `SubLevel` 1, requesting the retained publication, receive it without any further interception. Subscribers at a `SubLevel` greater than 1 are not sent the retained publication. As a result, the retained publication is not modified by the chain of intercepting subscribers a second time round.

Examples

The examples are code fragments that can be combined to build an intercepting subscriber. The code is written to be brief, rather than of production quality.

The preprocessor directives in Figure 87 on page 812 define the two properties to be extracted from the publication messages that are required by the `MQINQMP` MQI call.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
#define      MQPUBOPTIONS        (MQPTR)(char*) "MQPubOptions",\
                                 0,\
                                 12,\
                                 MQVS_NULL_TERMINATED,\
                                 MQCCSI_APPL
#define      MQTOPICSTRING       (MQPTR)(char*) "MQTopicString",\
                                 0,\
                                 13,\
                                 MQVS_NULL_TERMINATED,\
                                 MQCCSI_APPL
```

Figure 87. Preprocessor directives

Figure 88 on page 813 lists the declarations used in the code fragments. Except for the highlighted terms, the declarations are standard for an IBM MQ application.

The highlighted Put and Get options are initialized to pass all context. The highlighted `MQTOPICSTRING` and `MQPUBOPTIONS` are MQCHARV initializers for property names that are defined in the preprocessor directives. The names are passed to `MQINQMP`.

```
int main(int argc, char **argv) {
    MQLONG   Reason            = MQRC_NONE;
    MQLONG   CompCode          = MQCC_OK;
    MQHCONN  Hcon              = MQHC_UNUSABLE_HCONN;
    MQCHAR   QMName[49]        = " ";
    MQCMHO   CrtMsgHOpts       = {MQCMHO_DEFAULT};
    MQHMSG   Hmsg              = MQHM_NONE;
    MQMD     md                = {MQMD_DEFAULT};
    MQHOBJ   gHobj             = MQHO_NONE;
    MQOD     getOD             = {MQOD_DEFAULT};
    MQGMO    gmo               = {MQGMO_DEFAULT};
    MQLONG   GO_Options        = MQOO_INPUT_AS_Q_DEF
                               | MQOO_FAIL_IF_QUIESCING
                               | MQOO_SAVE_ALL_CONTEXT;
    MQLONG   GC_Options        = MQCO_DELETE_PURGE;
    MQHOBJ   Hsub              = MQHO_NONE;
    MQSD     sd                = {MQSD_DEFAULT};
    MQLONG   SC_Options        = MQCO_NONE;
    MQHOBJ   pHobj             = MQHO_NONE;
    MQOD     putOD             = {MQOD_DEFAULT};
    MQLONG   PO_Options        = MQOO_OUTPUT
                               | MQOO_FAIL_IF_QUIESCING
                               | MQOO_PASS_ALL_CONTEXT;
    MQLONG   PC_Options        = MQCO_NONE;
    MQPMO    pmo               = {MQPMO_DEFAULT};
    MQIMPO   InqPropOpts       = {MQIMPO_DEFAULT};
    MQPD     PropDesc          = {MQPD_DEFAULT};
    MQLONG   Type              = MQTYPE_AS_SET;
    MQCHARV  TopStrProp        = {MQTOPICSTRING};
    MQCHARV  PubOptProp        = {MQPUBOPTIONS};
    MQLONG   DataLength        = 0;
    MQBYTE   buffer[256]       = "";
    MQLONG   buflen            = sizeof(buffer) - 1;
    MQLONG   messlen           = 0;
    char     TopStrBuf[256]    = "Initial value";
    int      i                 = 0;
```

Figure 88. Declarations

Initializations that are not easily performed in declarations are shown in Figure 89 on page 814. The highlighted values require explanation.

SYSTEM.NDURABLE.MODEL.QUEUE
: In this example, instead of using MQSUB to open a managed non-durable subscription, the model queue, SYSTEM.NDURABLE.MODEL.QUEUE, is used to create a temporary dynamic queue. Its handle is passed to MQSUB. By opening the queue directly you are able to save all message context and set the subscription option, MQSO_SET_CORREL_ID.

MQGMO_CURRENT_VERSION
: It is important to use the current version of most of the IBM MQ structures. Fields such as gmo.MsgHandle are only available in the latest version of the control structures.

MQGMO_PROPERTIES_IN_HANDLE
: The topic string and put message options set in the original publication are to be retrieved by the intercepting subscriber using message properties. An alternative would be to read the **MQRFH2** structure in the message directly.

MQSO_SET_CORREL_ID
: Use MQSO_SET_CORREL_ID in combination with,

```
memcpy(sd.SubCorrelId, MQCI_NONE, sizeof(sd.SubCorrelId));
```

The effect of these options is to pass on the correlation identifier. The correlation identifier set by the original publisher is placed in the correlation identifier field of the publication that is received by the intercepting subscriber. Each intercepting subscriber passes on the same correlation identifier. The final subscriber then has the option of receiving the same correlation identifier.

Note: If the publication is passed through a publish/subscribe hierarchy, the correlation identifier is never retained.

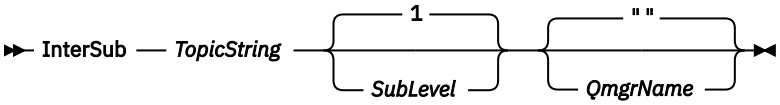
MQPRI_PRIORITY_AS_PUBLISHED
The publication is placed on the publication queue with the same message priority as it was published with.

```
strncpy(getOD.ObjectName,    "SYSTEM.NDURABLE.MODEL.QUEUE",
        sizeof(getOD.ObjectName));
gmo.Version                = MQGMO_VERSION_4;
gmo.Options                = MQGMO_WAIT
                           | MQGMO_PROPERTIES_IN_HANDLE
                           | MQGMO_CONVERT;
gmo.WaitInterval           = 30000;
sd.Options                 = MQSO_CREATE
                           | MQSO_FAIL_IF_QUIESCING
                           | MQSO_SET_CORREL_ID;
sd.PubPriority             = MQPRI_PRIORITY_AS_PUBLISHED;
sd.Version                 = MQSD_VERSION_1;
memcpy(sd.SubCorrelId, MQCI_NONE, sizeof(sd.SubCorrelId));
putOD.ObjectType             = MQOT_TOPIC;
putOD.ObjectString.VSPtr     = &TopStrBuf;
putOD.ObjectString.VSBufSize = sizeof(TopStrBuf);
putOD.ObjectString.VSLength  = MQVS_NULL_TERMINATED;
putOD.ObjectString.VSCCSID   = MQCCSI_APPL;
putOD.Version                = MQOD_VERSION_4;
pmo.Version                = MQPMO_VERSION_3;
```

Figure 89. Initializations

Figure 90 on page 815 shows the code fragment to read command-line parameters, complete the initialization, and create the intercepting subscription.

Run the program with the command,



To make error handling as unobtrusive as possible, the reason code from each MQI call is stored in a different array element. After each call the completion code is tested, and if the value is MQCC_FAIL, control exits the `do { } while(0)` code block.

The two noteworthy lines of code are,

pmo.PubLevel = sd.SubLevel - 1;
Sets the publication level for the republished message to one less than the subscription level of the intercepting subscriber.

gmo.MsgHandle = Hmsg;
Provides a message handle for MQGET to return the message properties.

```
do {
   printf("Intercepting subscriber start\n");
   if (argc < 2) {
       printf("Required parameter missing - topic string\n");
       exit(99);
   } else {
       sd.ObjectString.VSPtr    = argv[1];
       sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
       printf("TopicString = %s\n", sd.ObjectString.VSPtr);
   }
   if (argc > 2) {
       sd.SubLevel  = atoi(argv[2]);
       pmo.PubLevel = sd.SubLevel - 1;
       printf("SubLevel is %d, PubLevel is %d\n", sd.SubLevel, pmo.PubLevel);
   }
   if (argc > 3)
       strncpy(QMName, argv[3], sizeof(QMName));
   MQCONN(QMName, &Hcon, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   MQOPEN(Hcon, &getOD, GO_Options, &gHobj, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   MQSUB(Hcon, &sd, &gHobj, &Hsub, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   MQCRTMH(Hcon, &CrtMsgHOpts, &Hmsg, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   gmo.MsgHandle = Hmsg;
```

Figure 90. Preparing to intercept publications

The main code fragment, Figure 91 on page 816, gets messages from the publication queue. It queries the message properties and republishes the messages using the topic string, and the original **MQPMO**. `option` properties of the publication.

In this example, no transformation is performed on the publication. The topic string of the republished publication always matches the topic string the intercepting subscriber subscribed on. If the intercepting subscriber is responsible for intercepting multiple subscriptions sent to the same publication queue, it might be necessary to query the topic string to distinguish publications that match different subscriptions.

The calls to `MQINQMP` are highlighted. The topic string and publication put message options properties are written directly into the output control structures. The only reason for altering the MQCHARV length field of `putOD.ObjectString` from an explicit length to a null terminated string is to use `printf` to output the string.

```
while (CompCode != MQCC_FAILED) {
   memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
   memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
   md.Encoding       = MQENC_NATIVE;
   md.CodedCharSetId = MQCCSI_Q_MGR;
   printf("MQGET : %d seconds wait time\n", gmo.WaitInterval/1000);
   MQGET(Hcon, gHobj, &md, &gmo, buflen, buffer, &messlen,
       &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   buffer[messlen] = '\0';
   MQINQMP(Hcon, Hmsg,    &InqPropOpts, &TopStrProp, &PropDesc, &Type,
       putOD.ObjectString.VSBufSize, putOD.ObjectString.VSPtr,
       &(putOD.ObjectString.VSLength), &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   memset((void *)((MQLONG)(putOD.ObjectString.VSPtr)
       + putOD.ObjectString.VSLength),'\0',1);
   putOD.ObjectString.VSLength = MQVS_NULL_TERMINATED;
   MQINQMP(Hcon, Hmsg, &InqPropOpts, &PubOptProp, &PropDesc, &Type,
       sizeof(pmo.Options), &(pmo.Options), &DataLength,
       &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   MQOPEN(Hcon, &putOD, PO_Options, &pHobj, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   printf("Republish message <%s> on topic <%s> with options %d\n",
       buffer, putOD.ObjectString.VSPtr, pmo.Options);
   MQPUT(Hcon, pHobj, &md, &pmo, messlen, buffer, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
       break;
   MQCLOSE(Hcon, &pHobj, PC_Options, &CompCode, &Reason);
   if (CompCode == MQCC_FAILED)
   break;
}
```

Figure 91. Intercept publication and republish

The final code fragment is shown in Figure 92 on page 816.

```
    } while (0);
    if (CompCode == MQCC_FAILED && Reason != MQRC_NO_MSG_AVAILABLE)
        printf("MQI Call failed with reason code %d\n", Reason);
    if (Hsub!= MQHO_NONE)
        MQCLOSE(Hcon, &Hsub, SC_Options, &CompCode, &Reason);
    if (Hcon!= MQHC_UNUSABLE_HCONN)
        MQDISC(&Hcon, &CompCode, &Reason);
}
```

Figure 92. Completion

Intercepting publications and distributed publish/subscribe
Follow a simple pattern when you deploy intercepting subscribers or Publish exits to a distributed publish/subscribe topology. Deploy intercepting subscribers on the same queue managers as publishers, and Publish exits on the same queue managers as final subscribers.

Figure 93 on page 817 shows two queue managers connected in a publish subscribe cluster. A publisher creates a publication to a cluster topic at publication level 9. The numbered arrows show the sequence of steps taken by the publication as it flows to subscribers to the cluster topic. The publication is intercepted by the subscriber with `Sublevel` 9 and republished with `Publevel` 8. It is intercepted again by a subscriber at `Sublevel` 8. The subscriber republishes at `Publevel` 7. The proxy subscriber provided by the queue manager forwards the publication to queue manager B, where a Publish exit has been deployed in addition to a final subscriber. The publication is processed by the Publish exit before it is finally received by the final subscriber at `Sublevel` 1. The intercepting subscribers and the publish exit are shown with broken outlines.

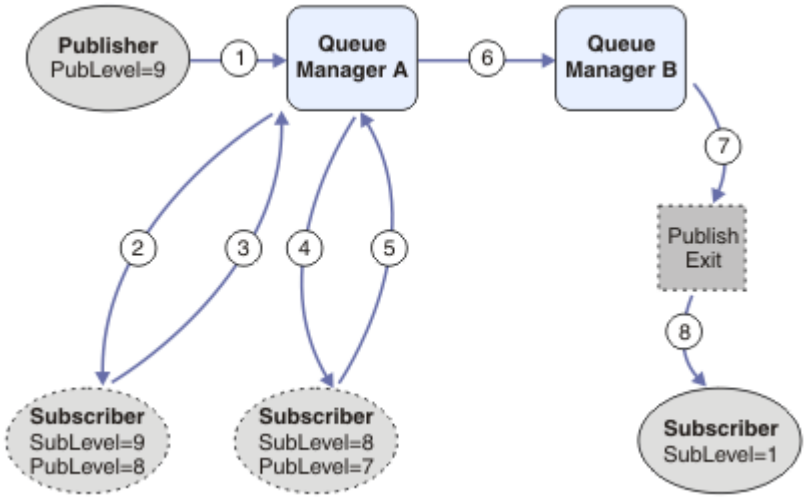


Figure 93. Interception and Publish exit in a cluster

The objective of the simple pattern is for every subscriber receiving a publication to receive the identical publication. The publication goes through the same sequence of transformations regardless of where the subscriber is connected. You probably want to avoid having the sequence of transformations varying, depending on where the publishers or final subscribers are connected. An reasonable exception would be to tailor the publication finally delivered to each individual subscriber. Use the Publish exit to do customize the publication based on the queue to which the publication is finally delivered.

You must consider carefully where to deploy intercepting subscribers and Publish exits in a distributed publish/subscribe topology. The straightforward pattern deploys intercepting subscribers to the same queue manager as the publishers, and Publish exits to the same queue managers as the final subscribers.

Anti-pattern

Figure 94 on page 818 shows how matters can go awry, if you do not follow a simple pattern.
To complicate the deployment, a final subscriber is added to queue manager A and two additional intercepting subscribers are added to queue manager B.

The publication is forwarded to queue manager B at `PubLevel` 7, where it is intercepted by a subscriber at `SubLevel` 5 before being consumed by the final subscriber at `SubLevel` 1. The Publish exit intercepts the publication before it is passed to both the intercepting consumer and the final consumer at queue manager B. The publication reaches the final subscriber on queue manager A without being processed by the Publish exit.

In a publish/subscribe topology, proxy subscribers subscribe at `SubLevel` 1, and pass on the `PubLevel` set by the last intercepting subscriber. In Figure 94 on page 818, the result is that the publication is not intercepted by the subscriber using `SubLevel` 9 at queue manager B.

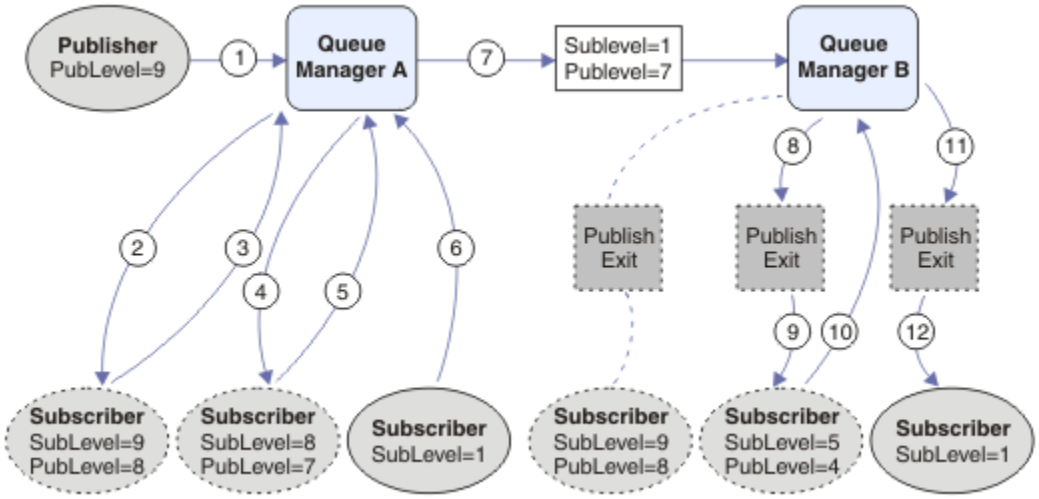


Figure 94. Complex deployment of intercepting subscribers

Publishing options

Several options are available that control the way messages are published.

Withholding reply-to information from subscribers

If you do not want subscribers to be able to reply to publications they receive, it is possible to withhold information in the ReplyToQ and ReplyToQmgr fields of the MQMD by using the MQPMO_SUPPRESS_REPLYTO put-message option. If this option is used, the queue manager removes that information from the MQMD when it receives the publication before forwarding it to any subscribers.

This option cannot be used in combination with a report option that needs a ReplyToQ, if this is attempted the call with fail with MQRC_MISSING_REPLY_TO_Q.

Publication level

Using publication levels is a way of controlling which subscribers receive the publication. The publication level denotes the level of subscription targeted by the publication. Only subscriptions with the highest subscription level less than or equal to the publication's publication level, will receive the publication. This value must be in the range zero to nine; zero is the lowest publication level. The initial value of this field is 9. One of the uses of publication and subscription levels is to intercept publications.

Checking if a publication is not delivered to any subscribers

To check if a publication has not been delivered to any subscribers, use the MQPMO_WARN_IF_NO_SUBS_MATCHED put-message option with the MQPUT call. If a completion code of MQCC_WARNING and a reason code MQRC_NO_SUBS_MATCHED are returned by the put operation, the publication was not delivered to any subscriptions. If the MQPMO_RETAIN option is specified on the put operation, the message is retained and delivered to any subsequently defined matching subscription. In a distributed publish/subscribe system, the MQRC_NO_SUBS_MATCHED reason code is returned only if there are no proxy subscriptions registered for the topic on the queue manager.

Subscription options

Several options are available that control the way message subscriptions are handled.

Message persistence

Queue managers maintain the persistence of the publications they forward to subscribers as set by the publisher. The publisher sets the persistence to be one of the following options:

0
Nonpersistent

1
Persistent

2
Persistence as queue/topic definition

For publish/subscribe, the publisher resolves the topic object and **topicString** to a resolved topic object. If the publisher specifies Persistence as queue/topic definition, then the default persistence from the resolved topic object is set for the publication.

Retained publications

To control when retained publications are received, subscribers can use two subscription options:

Publish on request only, MQSO_PUBLICATIONS_ON_REQUEST

If you want a subscriber to have control of when it receives publications you can use the MQSO_PUBLICATIONS_ON_REQUEST subscription option. A subscriber can then control when it receives publications by using the MQSUBRQ call (specifying the Hsub handle that was returned from the original MQSUB call) to request that it is sent a topic's retained publication. Subscribers using the MQSO_PUBLICATIONS_ON_REQUEST subscription option, do not receive any non-retained publications.

If you specify MQSO_PUBLICATIONS_ON_REQUEST you must use MQSUBRQ to retrieve any publication. If you do not use MQSO_PUBLICATIONS_ON_REQUEST you get messages as they are published.

If a subscriber uses the MQSUBRQ call and uses wildcards in the subscription's topic, the subscription might match multiple topics or nodes on a topic tree, all of which with retained messages (if any exist) will be sent to the subscriber.

This option can be particularly helpful when used with durable subscriptions because a queue manager will continue to send publications to a subscriber if it subscribed durably even if that subscriber application is not running. This could lead to a buildup of messages on the subscriber queue. This build up can be avoided if the subscriber registers using the MQSO_PUBLICATIONS_ON_REQUEST option. Alternatively, you can use non-durable subscriptions if appropriate to your application to avoid a build up of unwanted messages.

If a subscription is durable and a publisher uses retained publications the subscriber application can use the MQSUBRQ call to refresh its state information after a restart. The subscriber must then refresh its state periodically using the MQSUBRQ call.

No publications will be sent as a result of the MQSUB call using this option. A durable subscription that has been resumed following disconnection will use the MQSO_PUBLICATIONS_ON_REQUEST option if the original subscription was configured to use this option.

New publications only, MQSO_NEW_PUBLICATIONS_ONLY

If a retained publication exists on a topic, any subscribers that make a subscription after the publication was made will receive a copy of that publication. If a subscriber does not want to receive any publications that were made earlier than the subscription being made, the subscriber can use the MQSO_NEW_PUBLICATIONS_ONLY subscription option.

Grouping subscriptions

Consider grouping subscriptions if you have set up a queue to receive publications and have a number of overlapping subscriptions feeding publications to the same queue. This situation is similar to the example in Overlapping subscriptions.

You can avoid receiving duplicate publications by setting the option MQSO_GROUP_SUB when you subscribe to a topic. The result is that when more than one subscription in the group matches the topic of a publication, only one subscription is responsible for placing the publication on the queue. The other subscriptions that matched the publication topic are ignored.

The subscription responsible for placing the publication on the queue is chosen on the basis that it has the longest matching topic string, before encountering any wildcards. It can be thought of as the closest matching subscription. Its properties are propagated to the publication, including whether it has the MQSO_NOT_OWN_PUBS property. If it does, no publication is delivered to the queue, even though other matching subscriptions might not have the MQSO_NOT_OWN_PUBS property.

You cannot place all your subscriptions in a single group to eliminate duplicate publications. Grouped subscriptions must fulfill these conditions:

1. None of the subscriptions are managed.
2. A group of subscriptions deliver publications to the same queue.
3. Each subscription must be at the same subscription level.
4. The publication message for each subscription in the group has the same correlation identifier.

   To ensure each subscription results in a publication message with the same correlation identifier, set MQSO_SET_CORREL_ID to create your own correlation identifier in the publication, and set the same value in the **SubCorrelId** field in each subscription. Do not set **SubCorrelId** to the value MQCI_NONE.

See ../refdev/q100080_.dita#q100080_/mqso_group_sub for more information.

Inquiring about and setting object attributes

Attributes are the properties that define the characteristics of an IBM MQ object.

They affect the way that a queue manager processes an object. The attributes of each type of IBM MQ object are described in detail in Attributes of objects.

Some attributes are set when the object is defined, and can be changed only by using the IBM 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 one of the following resources:

- ALW The MQSC facility, which is described in MQSC commands.
- IBM i The CHGMQMx CL commands, which are described in CL commands reference for IBM i, or the MQSC facility.
- z/OS The ALTER operator commands, or the DEFINE commands with the REPLACE option, which are described in MQSC commands.

Note: The names of the attributes of objects are shown in this documentation in the form that you use them with the MQINQ and MQSET calls. When you use IBM 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 topic links.

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:

Table 127. Prefixes for selector names

Prefix	Description
MQCA_	These selectors refer to attributes that contain character data (for example, the name of a queue).

Table 127. Prefixes for selector names (continued)	
Prefix	**Description**
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 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 709 and "Opening and closing objects" on page 716.

Use the following links to find out more about getting inquiring about and setting object attributes:

- "Inquiring about the attributes of an object" on page 821
- "Some cases where the MQINQ call fails" on page 822
- "Setting queue attributes" on page 823

Related concepts

"The Message Queue Interface overview" on page 697
Learn about the Message Queue Interface (MQI) components.

"Connecting to and disconnecting from a queue manager" on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Opening and closing objects" on page 716
This information provides an insight into opening and closing IBM MQ objects.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Getting messages from a queue" on page 741
Use this information to learn about getting messages from a queue.

"Committing and backing out units of work" on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Working with the MQI and clusters" on page 852
There are special options on calls and return codes that relate to clustering.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

"IMS and IMS bridge applications on IBM MQ for z/OS" on page 66
This information helps you to write IMS applications using IBM MQ.

Inquiring about the attributes of an object

Use the MQINQ call to inquire about the attributes of any type of IBM MQ.

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 with a value that 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 or MQOO_BIND_ON_GROUP was used or MQOO_BIND_AS_Q_DEF was used and the **DefBind** queue attribute was MQBND_BIND_NOT_FIXED. See "MQOPEN and clusters" on page 853 and MQOPEN 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 MQINQ.

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 IBM 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 attributes upon which 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 attributes upon which 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

Use this information to learn how to set queue attributes using the MQSET call.

You can set only the following queue attributes using the MQSET call:

- *InhibitGet* (but not for remote queues)
- Multi *DistList*
- *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 IBM MQ objects other than locally defined queues.

For more details about the MQSET call, see MQSET.

Committing and backing out units of work

This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

The following terms 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 IBM MQ applications" on page 825.

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 is not permanent, it can back out the unit of work. When a program performs a backout, IBM 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, IBM 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 MAXUMSGS attribute of the ALTER QMGR command.

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 IBM 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, IBM 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 abnormally end, 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.

Use the following links to find out more about committing and backing out units of work:

- “Syncpoint considerations in IBM MQ applications” on page 825
- z/OS “Syncpoints in IBM MQ for z/OS applications” on page 826
- IBM i “Syncpoints in CICS for IBM i applications” on page 828
- “Syncpoints in IBM MQ for Multiplatforms” on page 828
- IBM i “Interfaces to the IBM i external syncpoint manager” on page 832

Related concepts

“The Message Queue Interface overview” on page 697
Learn about the Message Queue Interface (MQI) components.

“Connecting to and disconnecting from a queue manager” on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

“Opening and closing objects” on page 716
This information provides an insight into opening and closing IBM MQ objects.

“Putting messages on a queue” on page 727
Use this information to learn how to put messages on a queue.

“Getting messages from a queue” on page 741
Use this information to learn about getting messages from a queue.

“Inquiring about and setting object attributes” on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

“Starting IBM MQ applications using triggers” on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

“Working with the MQI and clusters” on page 852
There are special options on calls and return codes that relate to clustering.

“Using and writing applications on IBM MQ for z/OS” on page 857
IBM 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.

“IMS and IMS bridge applications on IBM MQ for z/OS” on page 66

This information helps you to write IMS applications using IBM MQ.

Syncpoint considerations in IBM MQ applications

Use this information to learn about using syncpoints in IBM MQ applications.

Two-phase commit is supported by the following environments:

- Multi IBM MQ for Multiplatforms
- z/OS CICS Transaction Server for z/OS
- z/OS TXSeries
- z/OS IMS/ESA®
- z/OS z/OS batch with RRS
- Other external coordinators using the X/Open XA interface

Single-phase commit is supported by the following environments:

- Multi IBM MQ for Multiplatforms
- z/OS z/OS batch

For further information about external interfaces, see "Interfaces to external syncpoint managers on Multiplatforms" on page 831, 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 IBM MQ can be brought within the scope of a unit of work, managed by the transaction manager.

Samples shipped with IBM MQ show IBM MQ coordinating XA-compliant databases. For further information about these samples, see "Using the IBM MQ sample procedural programs" on page 1018.

In your IBM 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:

- Multi The syncpoint control default is NO.
- z/OS The syncpoint control default is YES.

When an MQPUT1 call is issued with MQPMO_SYNCPOINT, the default behavior changes, so that the put operation is completed asynchronously. This might cause a change in the behavior of some applications that rely on certain fields in the MQOD and MQMD structures being returned, but which now contain undefined values. An application can specify MQPMO_SYNC_RESPONSE to ensure that the put operation is performed synchronously and that all of the appropriate field values are completed.

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 MQBACK and then, if appropriate, try the entire transaction again. 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*.

If a program issues the MQDISC call while there are uncommitted requests, an implicit syncpoint occurs (except on z/OS batch with RRS). If the program ends abnormally, an implicit backout occurs.

z/OS On z/OS, an implicit syncpoint also occurs if the program ends normally without first calling MQDISC. The program is deemed to have ended normally if the TCB connected to MQ ends normally. When running under z/OS UNIX System Services and Language Environment (LE), default condition handling is invoked for abends or signals. The LE condition handlers process the error condition and the TCB ends normally. Under these conditions MQ commits the unit of work. For more information, see Introduction to Language Environment Condition Handling.

z/OS For IBM MQ for z/OS programs, you can use the MQGMO_MARK_SKIP_BACKOUT option to specify that a message must 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 771.

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.

z/OS *Syncpoints in IBM MQ for z/OS applications*

This topic explains how to use syncpoints in transaction manager (CICS and IMS) and batch applications.

z/OS *Syncpoints in CICS Transaction Server for z/OS 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.

z/OS *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 IMS documentation.

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 batch or non-message driven BMP environment). This is because a different user could initiate the next unit of work and IBM MQ security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made.

However, in a Wait-for-Input (WFI) or pseudo Wait-for-Input (PWFI) environment IMS does not notify IBM MQ to close the handles until either the next message arrives or a QC status code is returned to the application. If the application is waiting in the IMS region and any of these handles belong to triggered queues, triggering will not occur because the queues are open. For this reason applications running in a WFI or PWFI environment should explicitly MQCLOSE the queue handles before doing the GU to the IOPCB for the next message.

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.

z/OS *Syncpoints in z/OS batch applications*

For batch applications, you can use the IBM MQ syncpoint management calls: MQCMIT and MQBACK. For compatibility with earlier versions, 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 IBM MQ and Db2, within a single unit of work you can use RRS. For further information see "Transaction management and recoverable resource manager services" on page 827.

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 MQCMIT.

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 MQBACK.

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 IBM 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.

IBM MQ for z/OS RRS participant support enables IBM MQ applications in the batch, TSO, and Db2 stored procedure environments to update both IBM MQ and non-IBM MQ resources (for example, Db2) within a single logical unit of work. For information about RRS participant support, see *z/OS MVS Programming: Resource Recovery*.

Your IBM MQ application can use either MQCMIT and MQBACK or the equivalent RRS calls, SRRCMIT and SRRBACK. See "The RRS batch adapter" on page 859 for more information.

RRS availability

If RRS is not active on your z/OS system, any IBM 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 IBM 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 IBM MQ calls, and is linked with a non-RRS stub, the MQCONN or MQCONNX call returns MQRC_ENVIRONMENT_ERROR, unless it is the first IBM MQ call executed since the stored procedure address space started.
- If your Db2 stored procedure contains IBM MQ calls and is linked with a non-RRS stub, IBM 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 an IBM 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 IBM 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).

IBM i *Syncpoints in CICS for IBM i applications*

IBM MQ for IBM i participates in CICS for IBM i units of work. You can use the MQI within a CICS for IBM i application to put and get messages inside the current unit of work.

You can use the EXEC CICS SYNCPOINT command to establish a syncpoint that includes the IBM MQ for IBM i operations. To back out all changes up to the previous syncpoint, you can use the EXEC CICS SYNCPOINT ROLLBACK command.

If you use MQPUT, MQPUT1, or MQGET with the MQPMO_SYNCPOINT, or MQGMO_SYNCPOINT, option set in a CICS for IBM i application, you cannot log off CICS for IBM i until IBM MQ for IBM i 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 IBM i.

Multi *Syncpoints in IBM MQ for Multiplatforms*

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 IBM 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. IBM MQ can coordinate such units of work itself. They can also be coordinated by an external commitment controller. For example:

- Another transaction manager
- IBM i The IBM i commitment controller

For full integrity, use a two-phase commit procedure. Two-phase commit can be provided by XA-compliant transaction managers and databases. For example:

- TXSeries
- UDB
- IBM i the IBM i commitment controller

ALW IBM MQ products can coordinate global units of work using a two-phase commit process.

IBM i IBM MQ for IBM i can act as a resource manager for global units of work within a WebSphere Application Server environment, but cannot act as a transaction manager.

Implicit syncpoint

When putting persistent messages, IBM MQ is optimized for putting persistent messages under syncpoint. Multiple applications putting persistent messages to the same queue perform better if those applications use syncpoint. This is because there is less contention for the queue, if syncpoint is used to put persistent messages.

ImplSyncOpenOutput adds an implicit syncpoint when applications put persistent messages outside of syncpoint. This provides a performance improvement, without applications being aware of the implicit syncpoint.

Implicit syncpoint only provides a performance boost when there are multiple applications putting to the queue, because it reduces contention for the queue. So, **ImplSyncOpenOutput** specifies the minimum number of applications that have a queue open for output before an implicit syncpoint is added. The default value is 2. This means, that if you do not specify **ImplSyncOpenOutput**, implicit syncpoint is only added if multiple applications are putting to the queue.

See Tuning parameters for more information.

Local units of work on Multiplatforms
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 IBM 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.

ALW *Global units of work on AIX, Linux, and Windows*
Use global units of work when you also need to include updates to resources belonging to other resource managers. 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 in an IBM MQ MQI client environment.

Here, IBM 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 IBM 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 first 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 MQBEGIN.

External syncpoint coordination

This occurs when a syncpoint coordinator other than IBM MQ has been selected; for example, CICS, Encina, or Tuxedo.

In this situation, IBM MQ for AIX, Linux, and Windows systems 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 IBM MQ API calls in CICS programs, whether 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 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 transactional support, see Transactional support scenarios, and also TXSeries CICS documentation, for example, *TXSeries for Multiplatforms CICS Administration Guide for Open Systems*.

Multi *Implicit syncpoint on Multiplatforms*

Implicit syncpoint support enables persistent message puts outside of syncpoint.

When putting persistent messages, IBM MQ is optimized for putting persistent messages under syncpoint. Multiple applications concurrently putting persistent messages to the same queue typically perform better if those applications use syncpoint. This is because the locking strategy of IBM MQ is more efficient if syncpoint is used when putting persistent messages.

The **ImplSyncOpenOutput** parameter in the `qm.ini` file, controls whether an implicit syncpoint can be added when applications put persistent messages outside of syncpoint. This can provide a performance improvement, without applications being aware of the implicit syncpoint.

Implicit syncpoint only provides a performance boost when there are multiple applications concurrently putting to the queue, because it reduces lock contention. **ImplSyncOpenOutput** specifies the minimum number of applications that have a queue open for output before an implicit syncpoint can be added. The default value is *2*. This means, that if you do not explicitly specify **ImplSyncOpenOutput**, implicit syncpoint is only added if multiple applications are putting to the queue.

If you add an implicit syncpoint, statistics reflect that happening, and you might see a transaction output from **runmqsc display conn**.

Set **ImplSyncOpenOutput**=*OFF* if you never want an implicit syncpoint added.

See Tuning parameters for more information.

Interfaces to external syncpoint managers on Multiplatforms

IBM MQ for Multiplatforms supports 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.

- ALW The resource manager for IBM MQ on AIX, Linux, and Windows is named MQSeries_XA_RMI.
- IBM i For IBM i, the resource manager name 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, IBM MQ for Multiplatforms fulfills 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:

Table 128. Versions of XA Switch	
Version	**Description**
MQRMIXASwitch	Static XA resource management
MQRMIXASwitchDynamic	Dynamic XA resource management

For a list of the libraries containing this structure see The IBM MQ XA switch structure.

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 IBM 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 IBM 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 The CHANNEL, TRPTYPE, CONNAME, and QMNAME parameters of the xa_open string.

Restrictions

1. Global units of work are not allowed with a shared Hconn (as described in "Shared (thread independent) connections with MQCONNX" on page 713.
2. IBM i IBM MQ for IBM i does not support dynamic registration of XA resource managers.

   The only transaction manager supported is WebSphere Application Server.
3. Windows 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.

   Note: Note: A JMS client application (CLIENT JEE application) running in a JEE server does not have this restriction, so a single JEE server-managed transaction can coordinate multiple queue managers in the same transaction. However, a JMS server application, running in bindings mode, is still 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.

IBM i *Interfaces to the IBM i external syncpoint manager*

IBM MQ for IBM i can use native IBM i commitment control as an external syncpoint coordinator.

Thread-independent (shared) connections are not allowed with commitment control. See the *IBM i Programming: Backup and Recovery Guide, SC21-8079* for more information about the commitment control capabilities of IBM i.

To start the IBM i commitment control facilities, use the STRCMTCTL system command. To end commitment control, use the ENDCMTCTL system command.

Note: The default value of *Commitment definition scope* is *ACTGRP. This must be defined as *JOB for IBM MQ for IBM i. For example:

```
STRCMTCTL LCKLVL(*ALL) CMTSCOPE(*JOB)
```

IBM MQ for IBM i can also perform local units of work containing only updates to IBM MQ resources. The choice between local units of work and participation in global units of work coordinated by IBM i 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, IBM MQ starts a local unit of work and all further units of work for this connection to IBM MQ also use local units of work, regardless of whether commitment control is then started. To commit a local unit of work, use MQCMIT. To back out a local unit of work, use MQBACK. The IBM i commit and rollback calls such as the CL command COMMIT have no effect on IBM MQ local units of work.

If you want to use IBM MQ for IBM i with native IBM i commitment control as an external syncpoint coordinator, ensure that any job with commitment control is active and that you are using IBM 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, IBM MQ for IBM i adds itself as an API commitment resource to the commitment definition. This is typically the first such call in a job. While there are any API commitment resources registered under a particular commitment definition, you cannot end commitment control for that definition.

IBM MQ for IBM i removes its registration as an API commitment resource when you disconnect from the queue manager, if there are no pending MQI operations in the current unit of work.

If you disconnect from the queue manager while there are pending MQPUT, MQPUT1, or MQGET operations in the current unit of work, IBM MQ for IBM i remains registered as an API commitment resource so that it is notified of the next commit or rollback. When the next syncpoint is reached, IBM MQ for IBM i 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 IBM MQ for IBM i operations, and disconnect from the queue manager. Thus, using COMMIT or ROLLBACK commands before ENDCMTCTL enables end-commitment control to complete successfully.

When you use IBM i 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 IBM i commitment control as an external syncpoint coordinator with IBM MQ for IBM i, IBM i performs a two-phase commit protocol in which IBM 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 IBM 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 IBM MQ applications using triggers

Learn about triggers and how to start IBM MQ applications using triggers.

Some IBM 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.

IBM 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 Triggering channels.

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 reduces logging (resulting in 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. Typically 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 IBM MQ clients running on AIX, Linux, and Windows. An application running in a client environment is the same as one running in a full IBM 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 Attributes for process definitions 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 any platform other than Linux, 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 typically a message arriving on an application queue, but it can also occur at other times. For example, see “Conditions for a trigger event” on page 840.

IBM MQ has a range of options to allow you to control the conditions that cause a trigger event (see “Controlling trigger events” on page 844).

Trigger message

The queue manager creates a *trigger message* when it recognizes a trigger event. 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 851).

Initiation queue

An *initiation queue* is a local queue on which the queue manager puts trigger messages. Note that an initiation queue cannot be an alias queue or a model queue.

A queue manager can own more than one initiation queue, and each one is associated with one or more application queues.

z/OS A shared queue, a local queue accessible by queue managers in a queue sharing group, can be an initiation queue on IBM 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.

z/OS On z/OS, the channel initiator is typically started manually, or it can be done automatically when a queue manager starts by changing CSQINP2 in the queue manager startup JCL.

Multi On Multiplatforms, the channel initiator is started automatically when the queue manager starts, or it can be started manually with the **runmqchi** command.

For more information, see “Initiation queue processing by trigger monitors” on page 847.

To understand how triggering works, consider Figure 95 on page 836, which is an example of trigger type FIRST (MQTT_FIRST).

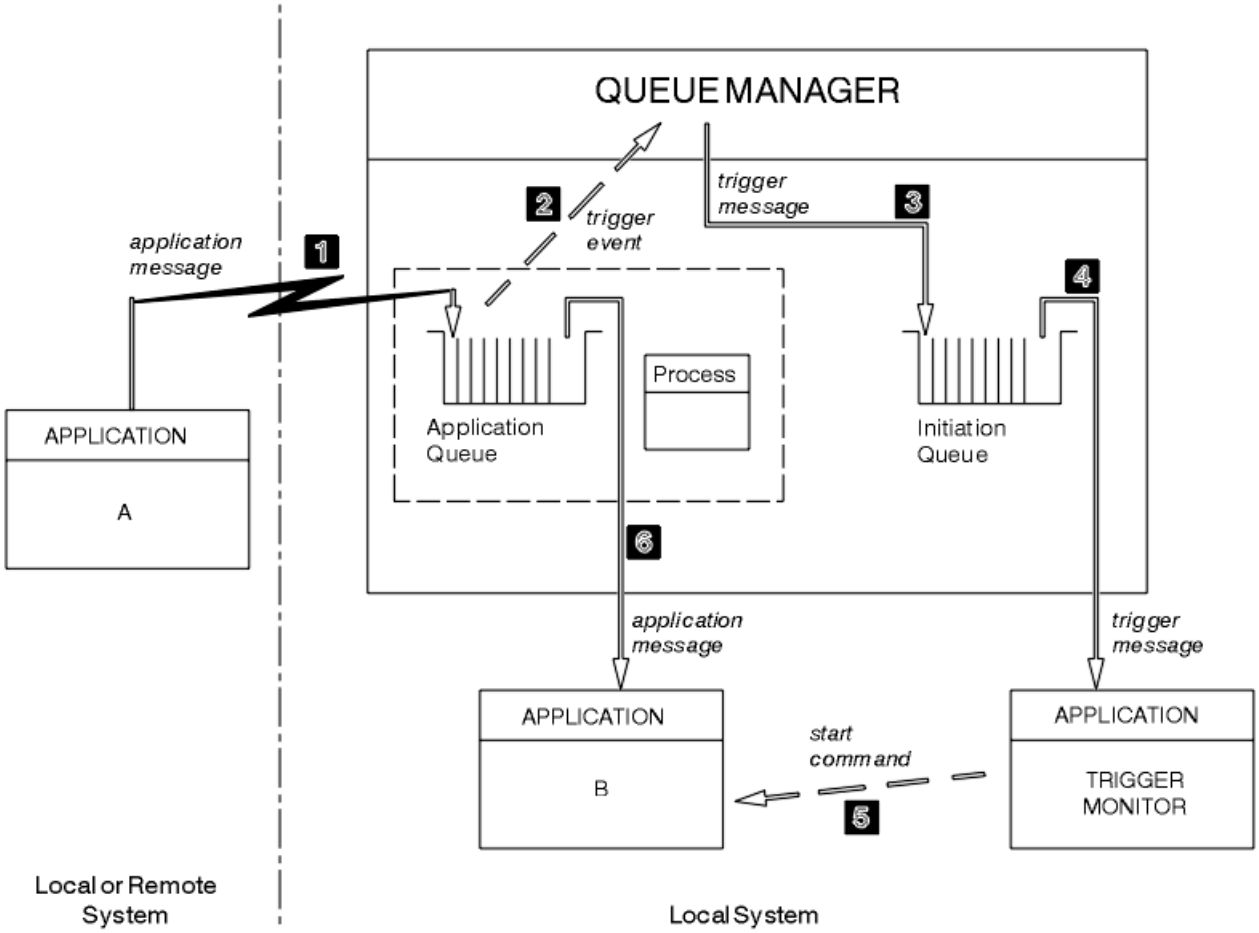


Figure 95. Flow of application and trigger messages

In Figure 95 on page 836, the sequence of events is:

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 occurs 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.
4. Triggered applications run under the user ID and group of the user who started the trigger monitor, the CICS user, or the user who started the queue manager.

So far, the relationship between the queues within triggering has been only on a one to one basis. Consider Figure 96 on page 837.

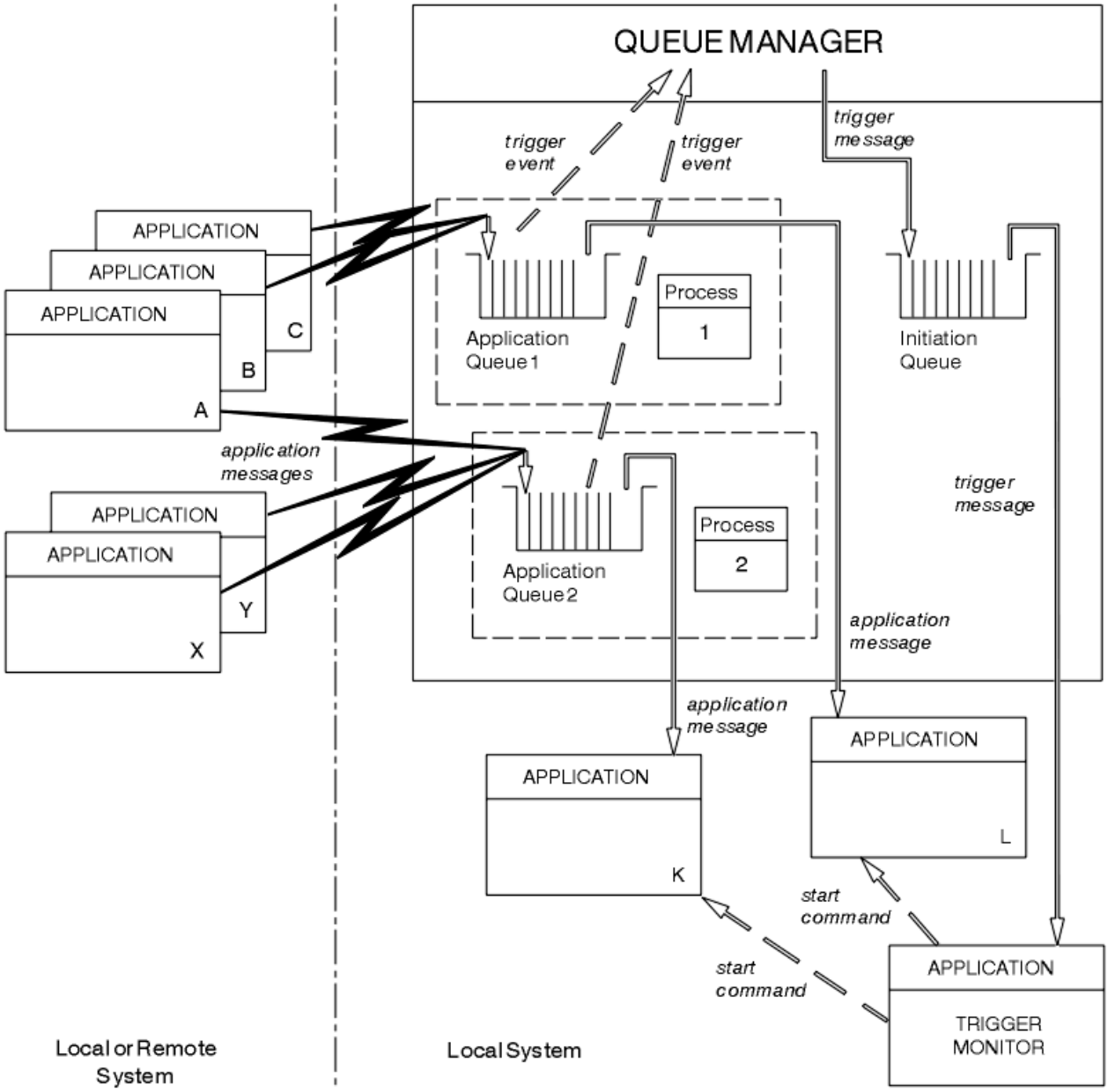


Figure 96. Relationship of queues within triggering

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.

Use the following links to find out more about starting IBM MQ applications using triggers:

- "Prerequisites for triggering" on page 838
- "Conditions for a trigger event" on page 840
- "Controlling trigger events" on page 844
- "Designing an application that uses triggered queues" on page 846

- “Initiation queue processing by trigger monitors” on page 847
- “Properties of trigger messages” on page 850
- “When triggering does not work” on page 852

Related concepts

“The Message Queue Interface overview” on page 697
Learn about the Message Queue Interface (MQI) components.

“Connecting to and disconnecting from a queue manager” on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

“Opening and closing objects” on page 716
This information provides an insight into opening and closing IBM MQ objects.

“Putting messages on a queue” on page 727
Use this information to learn how to put messages on a queue.

“Getting messages from a queue” on page 741
Use this information to learn about getting messages from a queue.

“Inquiring about and setting object attributes” on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

“Committing and backing out units of work” on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

“Working with the MQI and clusters” on page 852
There are special options on calls and return codes that relate to clustering.

“Using and writing applications on IBM MQ for z/OS” on page 857
IBM 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.

“IMS and IMS bridge applications on IBM MQ for z/OS” on page 66
This information helps you to write IMS applications using IBM MQ.

Prerequisites for triggering

Use this information to learn about the steps to take before using triggering.

Before your application can take advantage of triggering, complete the following steps:

1. Either:
   a. Create an initiation queue for your application queue. For example:

```
DEFINE QLOCAL (initiation.queue) REPLACE      +
       LIKE (SYSTEM.DEFAULT.INITIATION.QUEUE)      +
       DESCR ('initiation queue description')
```

   or

   b. Determine the name of a local queue that exists and can be used by your application (typically, this name 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)
```

IBM i Here is an extract from a CL program for IBM MQ for IBM i 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') +
         APPLICID ('PAYR') +
         APPLTYPE (CICS) +
         USERDATA ('Payroll data')
```

IBM i Here is an extract from a CL program for IBM MQ for IBM i 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
AIX, Linux, and Windows systems	Use DEFINE PROCESS or use SYSTEM.DEFAULT.PROCESS and modify using ALTER PROCESS
z/OS z/OS	Use DEFINE PROCESS (see sample code in step “3” on page 839), or use the operations and control panels.
IBM i IBM i	Use a CL program containing code as in step “3” on page 839.

4. Optional: 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 IBM 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
AIX, Linux, and Windows systems	ALTER QLOCAL
z/OS z/OS z/OS	ALTER QLOCAL
IBM i IBM i IBM i	CHGMQMQ

6. Start instances of the trigger monitors IBM i (or trigger servers in IBM MQ for IBM i) that are to serve the initiation queues you have defined. See “Initiation queue processing by trigger monitors” on page 847 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, see “Controlling trigger events” on page 844.

Conditions for a trigger event

The queue manager creates a trigger message when certain conditions are satisfied.

The following conditions cause the queue manager to create a trigger message:

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:

   - 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.
   - z/OS 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: z/OS

- 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 conditions specified in the queue definition.

  An example scenario affected by this condition is multiple queue managers QM1, QM2, and QM3 with a trigger monitor running for an application queue A. A message arrives on A satisfying the conditions for triggering, and a trigger message is generated on the initiation queue. The trigger monitor on QM1 gets the trigger message and triggers an application. The triggered application opens the application queue for shared input. From this point on the trigger conditions for application queue A are evaluated as *TriggerType* MQTT_FIRST, and *TriggerMsgPriority* zero on queue managers QM2 and QM3, until QM1 closes the application queue.
- 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.

  The change in how the trigger conditions are evaluated occurs when the triggered application opens the queue for input. In scenarios where there is only one trigger monitor running, other applications can have the same effect because they similarly open the application queue for input. It does not matter whether the application queue was opened by an application that is started by a trigger monitor, or by some other application; it is the fact that the queue is open for input on another queue manager that causes the change in trigger criteria.

5. z/OS On IBM 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 value of the **InhibitGet** queue attribute is MQQA_GET_ALLOWED).
   - Put requests must not be inhibited for the initiation queue (that is, the value of the **InhibitPut** queue attribute must be MQQA_PUT_ALLOWED).
   - The value of 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 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 840 through "10" on page 842 (excluding "3" on page 840) 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 trigger attribute of the queue 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 840), and conditions "6" on page 841 through "10" on page 842 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 44.
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 840), and conditions “4” on page 840 through “10” on page 842 (excluding “8” on page 842) 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 value of 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 840) on any of the queues for which this is the initiation queue, and conditions “4” on page 840 through “10” on page 842 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 value of 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 840) on the queue, and conditions “4” on page 840 through “10” on page 842, excluding “5” on page 841, 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 840) on any of the application queues for which this is the initiation queue, and conditions “4” on page 840 through “10” on page 842 (excluding “8” on page 842) 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” on page 844.

    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.

Notes:

1. From step “12” on page 842 (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 IBM 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, IBM 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 IBM MQ network.
3. z/OS For IBM MQ for z/OS, if a shared queue is setup for triggering and connection to the Coupling Facility hosting the shared queue is lost, a trigger event might be generated and a message put to the initiation queue. This can happen even when no message was put to the original shared queue setup for triggering. This is caused by the over-indication of bits by the IXLVECTR macro as documented in The List Notification Vector.

Controlling trigger events

You control trigger events using some of the attributes that define your application queue. This information also gives examples of using the trigger types: EVERY, FIRST, and DEPTH.

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 Attributes of objects.

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.

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. You must always process the queue until it is empty. See also “Special case of trigger type FIRST” on page 845.

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 *TriggerControl* 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 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 840.

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 might set a trigger of type EVERY on this reply-to queue so that each time a reply arrives, the reply might 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 days 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 might separate these items into four request messages, sending each to a separate destination. It might 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 typically 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.
- z/OS On IBM MQ for z/OS, application queues that name an open initiation queue are scanned periodically. If *TRIGINT* 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.)

- z/OS On IBM MQ for z/OS if you set *TRIGINT* 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 that are generated depends on the trigger interval. If the trigger interval is set to zero, the maximum number of messages are generated.

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 made available on the initiation queue when the UOW is resolved, 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.

   Created trigger messages are always made available, whether the unit of work is backed out or committed.
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 later, rely on your application being retriggered to process them. This method prevents applications being idle, and unnecessarily using resources.

Initiation queue processing by 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 correct actions on the correct 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.

ALW *Trigger monitors on AIX, Linux, and Windows systems*

This topic contains information about trigger monitors provided on AIX, Linux, 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 "Using the sample programs on Multiplatforms" on page 1018 for more information about 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 double 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 double 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 much 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 IBM MQ on the number of applications that can run.

Linux AIX To run the started application in the background on AIX and Linux, put an & at the end of the *EnvData* of the PROCESS definition.

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
```

Note: Windows Where a Windows path has spaces as a part of the path name, these should be enclosed in quotation marks (") 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 string enclosed in double quotation marks. START specifies that the first argument in quotation marks will be treated as the title of the new command. To ensure that Windows does not mistake the application path for a 'title' argument, add a title string enclosed in double quotation marks to the command before the application name.

The following trigger monitors are provided for the IBM MQ client:

runmqtmc
This is the same as runmqtrm except that it links with the IBM MQ MQI client libraries.

ALW *Trigger monitor 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.

This topic applies only to Windows, AIX and Linux x86-64 systems.

The trigger monitor 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 IBM 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 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 starts and stops, and when an error occurs. These messages are sent to the CSMT transient data queue.

Table 129. Available versions of the trigger monitor.

A table with two columns. the first colum lists the available versions of the trigger monitor and the second column shows which platforms each version is used for.

Version	Use
amqltmc0	TXSeries for: • AIX AIX • Linux Linux x86-64 systems
amqltmc4	Windows TXSeries for Windows 5.1
amqltmcc	Client bound version of the CICS trigger monitor
V 9.4.0 V 9.4.0 amqltmc064	64 bit TXSeries for Linux x86-64 systems
V 9.4.0 V 9.4.0 amqltmcc64	Client version of amqltmc064

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 perform the following actions:

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.

   z/OS For example, on 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. To name the process definition object, you can use the **DEFINE QLOCAL** or **ALTER QLOCAL** command.

   IBM i On IBM i, you can also use CRTMQMQ or CHGMQMQ.

For more information about the trigger monitor interface, see MQTMC2.

IBM i *Trigger monitors on IBM i*

On IBM i, instead of the **runmqtrm** control command, use the IBM MQ for IBM i CL command **STRMQMTRM**.

Use the STRMQMTRM command as follows:

```
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 IBM i job for the process that is to be started, but this means that there is additional processing 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 MQTMC2. Both of these samples are delivered in both source and executable forms.

Because these trigger monitors can invoke only native IBM i 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.

Here is the source of a sample CLP:

```
PGM PARM(&TMC2)
  DCL &TMC2 *CHAR LEN(800)
    ADDENVVAR ENVVAR(TM) VALUE(&TMC2)
    QSH  CMD('java_pgmname $TM')
    RMVENVVAR ENVVAR(TM)
ENDPGM
```

The following trigger monitor program is provided for the IBM MQ MQI client: RUNMQTMC

Call the RUNMQTMC as follows:

```
CALL PGM(QMQM/RUNMQTMC) PARM('-m' QMgrName '-q' InitQ)
```

Properties of trigger messages

The following topics describe some other properties of trigger messages.

- "Persistence and priority of trigger messages" on page 850
- "Queue manager restart and trigger messages" on page 851
- "Trigger messages and changes to object attributes" on page 851
- "Format of trigger messages" on page 851

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. If 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 IBM MQ for Multiplatforms, 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 the trigger monitor CKTI, supplied by CICS, the **ApplId** attribute of the process definition object is a CICS transaction identifier.

z/OS When you use CSQQTRMN supplied by IBM MQ for z/OS, the **ApplId** attribute of the process definition object is an 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 CICS-supplied trigger monitor (CKTI) or the IBM MQ for z/OS-supplied trigger monitor (CSQQTRMN) does 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 MQTM.

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; otherwise, 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 z/OS to the z/OS console or to the system operator or is 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 receives 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.

Working with the MQI and clusters

There are special options on calls and return codes that relate to clustering.

Use the following links to find out more about the options available on the calls and return codes for use with clusters:

- "MQOPEN and clusters" on page 853
- "MQPUT, MQPUT1 and clusters" on page 854
- "MQINQ and clusters" on page 854
- "MQSET and clusters" on page 855
- "Return codes" on page 855

Related concepts

"The Message Queue Interface overview" on page 697
Learn about the Message Queue Interface (MQI) components.

"Connecting to and disconnecting from a queue manager" on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Opening and closing objects" on page 716
This information provides an insight into opening and closing IBM MQ objects.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Getting messages from a queue" on page 741

Use this information to learn about getting messages from a queue.

"Inquiring about and setting object attributes" on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

"Committing and backing out units of work" on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

"IMS and IMS bridge applications on IBM MQ for z/OS" on page 66
This information helps you to write IMS applications using IBM MQ.

MQOPEN and clusters

The queue to which a message is put to, or read from, when a cluster queue is opened, depends on the MQOPEN call.

Selecting the target queue

If you do not provide a queue manager name in the object descriptor, MQOD, the queue manager selects the queue manager to send the message to. If you do provide a queue manager name in the object descriptor, then messages are always sent to the queue manager you have selected.

If the queue manager is selecting the target queue manager, the selection depends on the binding options, MQOO_BIND_* and if a local queue exists. If there is a local instance of the queue, it is always opened in preference to a remote instance, unless the CLWLUSEQ attribute is set to ANY. Otherwise, the selection depends on the binding options. Either MQOO_BIND_ON_OPEN or MQOO_BIND_ON_GROUP must be specified when using message groups with clusters to ensure that all messages in the group are processed at the same destination.

If the queue manager is selecting the target queue manager, it does so in a round-robin fashion, using the workload management algorithm; see Workload balancing in clusters.

When the workload balancing algorithm is used depends on the way the cluster queue is opened:

- MQOO_BIND_ON_OPEN - the algorithm is used once at the time the queue is opened by the application.
- MQOO_BIND_NOT_FIXED - the algorithm is used for every message put to the queue.
- MQOO_BIND_ON_GROUP - the algorithm is used once at the start of each message group.

MQOO_BIND_ON_OPEN
The MQOO_BIND_ON_OPEN option on the MQOPEN call specifies that the target queue manager is to be fixed. Use the MQOO_BIND_ON_OPEN option if there are multiple instances of the same queue within a cluster. All messages put to the queue specifying the object handle returned from the MQOPEN call are directed to the same queue manager.

- Use the MQOO_BIND_ON_OPEN option if messages have affinities. For example, if a batch of messages is all to be processed by the same queue manager, specify MQOO_BIND_ON_OPEN when you open the queue. IBM MQ fixes the queue manager and the route to be taken by all messages put to that queue.
- If MQOO_BIND_ON_OPEN option is specified, the queue must be reopened for a new instance of the queue to be selected.

MQOO_BIND_NOT_FIXED
The MQOO_BIND_NOT_FIXED option on the MQOPEN call specifies that the target queue manager is not fixed. Messages written to the queue specifying the object handle returned from the MQOPEN call are routed to a queue manager at MQPUT time on a message-by-message basis. Use the

MQOO_BIND_NOT_FIXED option if you do not want to force all your messages to be written to the same destination.

- Do not specify MQOO_BIND_NOT_FIXED and MQMF_SEGMENTATION_ALLOWED at the same time. If you do, the segments of your message might be delivered to different queue managers, scattered throughout the cluster.

MQOO_BIND_ON_GROUP
Allows an application to request that a group of messages is allocated to the same destination instance. This option is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

- Groups are only routed to a single destination when MQPMO_LOGICAL_ORDER is specified on the MQPUT. When MQOO_BIND_ON_GROUP is specified, but a message is not part of a logical group, BIND_NOT_FIXED behavior is used instead.

MQOO_BIND_AS_Q_DEF
If you do not specify either MQOO_BIND_ON_OPEN, MQOO_BIND_NOT_FIXED or MQOO_BIND_ON_GROUP, the default option is MQOO_BIND_AS_Q_DEF. Using MQOO_BIND_AS_Q_DEF causes the binding that is used for the queue handle to be taken from the DefBind queue attribute.

Relevance of MQOPEN options

The MQOPEN options MQOO_BROWSE, MQOO_INPUT_*, or MQOO_SET require a local instance of the cluster queue for MQOPEN to succeed.

The MQOPEN options MQOO_OUTPUT, MQOO_BIND_*, or MQOO_INQUIRE do not require a local instance of the cluster queue to succeed.

Resolved queue manager name

When a queue manager name is resolved at MQOPEN time, the resolved name is returned to the application. If the application tries to use this name on a subsequent MQOPEN call, it might find that it is not authorized to access the name.

MQPUT, MQPUT1 and clusters

If MQOO_BIND_NOT_FIXED is specified on an MQOPEN the workload management routines chooses which destination MQPUT or MQPUT1 select.

If MQOO_BIND_NOT_FIXED is specified on an MQOPEN call, each subsequent MQPUT call invokes the workload management routine to determine which queue manager to send the message to. The destination and route to be taken are selected on a message-by-message basis. The destination and route might change after the message has been put if conditions in the network change. The MQPUT1 call always operates as though MQOO_BIND_NOT_FIXED were in effect, that is, it always invokes the workload management routine.

When the workload management routine has selected a queue manager, the local queue manager completes the put operation. The message can be placed on different queues:

1. If the destination is the local instance of the queue, the message is placed on the local queue.
2. If the destination is a queue manager in a cluster, the message is placed on a cluster transmission queue.
3. If the destination is a queue manager outside a cluster, the message is placed on a transmission queue with the same name as the target queue manager.

If MQOO_BIND_ON_OPEN is specified on the MQOPEN call, MQPUT calls do not invoke the workload management routine because the destination and route have already been selected.

MQINQ and clusters

Which cluster queue is inquired upon depends upon the options you combine with MQOO_INQUIRE.

Before you can inquire on a queue, open it using the MQOPEN call and specify MQOO_INQUIRE.

To inquire on a cluster queue, use the MQOPEN call and combine other options with MQOO_INQUIRE. The attributes that can be inquired depend on whether there is a local instance of the cluster queue, and on how the queue is opened:

- Combining MQOO_BROWSE, MQOO_INPUT_*, or MQOO_SET with MQOO_INQUIRE requires a local instance of the cluster queue for the open to succeed. In this case you can inquire on all the attributes that are valid for local queues.
- Combining MQOO_OUTPUT with MQOO_INQUIRE, and specifying none of the preceding options, the instance opened is either:
  - The instance on the local queue manager, if there is one. In this case you can inquire on all the attributes that are valid for local queues.
  - An instance elsewhere in the cluster, if there is no local queue manager instance. In this case only the following attributes can be inquired on. The QType attribute has the value MQQT_CLUSTER in this case.
    - DefBind
    - DefPersistence
    - DefPriority
    - InhibitPut
    - QDesc
    - QName
    - QType

To inquire on the DefBind attribute of a cluster queue, use the MQINQ call with the selector MQIA_DEF_BIND. The value returned is either MQBND_BIND_ON_OPEN or MQBND_BIND_NOT_FIXED, or MQBND_BIND_ON_GROUP. Either MQBND_BIND_ON_OPEN or MQBND_BIND_ON_GROUP must be specified when using groups with clusters.

To inquire on the CLUSTER and CLUSNL attributes of the local instance of a queue, use the MQINQ call with the selector MQCA_CLUSTER_NAME or the selector MQCA_CLUSTER_NAMELIST.

Note: If you open a cluster queue without fixing the queue that MQOPEN has bound to, successive MQINQ calls might inquire on different instances of the cluster queue.

Related concepts

"MQOPEN option for cluster queue" on page 722

The binding used for the queue handle is taken from the **DefBind** queue attribute, which can take the value MQBND_BIND_ON_OPEN, MQBND_BIND_NOT_FIXED, or MQBND_BIND_ON_GROUP.

MQSET and clusters

The MQOPEN option MQOO_SET option requires there to be a local instance of a cluster queue for MQSET to succeed.

You cannot use the MQSET call to set the attributes of a queue elsewhere in the cluster.

You can open a local alias or remote queue defined with the cluster attribute and use the MQSET call. You can set the attributes of the local alias or remote queue. It does not matter if the target queue is a cluster queue defined on a different queue manager.

Return codes

Return codes specific to clusters

MQRC_CLUSTER_EXIT_ERROR (2266 X'8DA')

An MQOPEN, MQPUT, or MQPUT1 call is issued to open a cluster queue or put a message on it. The cluster workload exit, defined by the ClusterWorkloadExit attribute of a queue manager, fails unexpectedly or does not respond in time.

z/OS A message is written to the system log on IBM MQ for z/OS giving more information about this error.

Subsequent MQOPEN, MQPUT, and MQPUT1 calls for this queue handle are processed as though the ClusterWorkloadExit attribute were blank.

MQRC_CLUSTER_EXIT_LOAD_ERROR (2267 X'8DB')

z/OS On z/OS, the cluster workload exit cannot be loaded.

A message is written to the system log and processing continues as though the ClusterWorkloadExit attribute is blank.

Multi On Multiplatforms, an MQCONN or MQCONNX call is issued to connect to a queue manager. The call fails because the cluster workload exit, defined by the queue manager ClusterWorkloadExit attribute of the queue manager, cannot be loaded.

MQRC_CLUSTER_PUT_INHIBITED (2268 X'8DC')

An MQOPEN call with the MQOO_OUTPUT and MQOO_BIND_ON_OPEN options in effect is issued for a cluster queue. All the instances of the queue in the cluster are currently put-inhibited by having the InhibitPut attribute set to MQQA_PUT_INHIBITED. Because there are no queue instances available to receive messages, the MQOPEN call fails.

This reason code occurs only when both of the following statements are true:

- There is no local instance of the queue. If there is a local instance, the MQOPEN call succeeds, even if the local instance is put-inhibited.
- There is no cluster workload exit for the queue, or there is a cluster workload exit but it does not choose a queue instance. (If the cluster workload exit chooses a queue instance, the MQOPEN call succeeds, even if that instance is put-inhibited.)

If the MQOO_BIND_NOT_FIXED option is specified on the MQOPEN call, the call can succeed even if all the queues in the cluster are put-inhibited. However, a subsequent MQPUT call might fail if all the queues are still put-inhibited at the time of that call.

MQRC_CLUSTER_RESOLUTION_ERROR (2189 X'88D')

1. An MQOPEN, MQPUT, or MQPUT1 call is issued to open a cluster queue or put a message on it. The queue definition cannot be resolved correctly because a response is required from the full repository queue manager but none is available.
2. An MQOPEN, MQPUT, MQPUT1 or MQSUB call is issued for a topic object specifying PUBSCOPE (ALL) or SUBSCOPE (ALL). The cluster topic definition cannot be resolved correctly because a response is required from the full repository queue manager but none is available.

MQRC_CLUSTER_RESOURCE_ERROR (2269 X'8DD')

An MQOPEN, MQPUT, or MQPUT1 call is issued for a cluster queue. An error occurs while trying to use a resource required for clustering.

MQRC_NO_DESTINATIONS_AVAILABLE (2270 X'8DE')

An MQPUT or MQPUT1 call is issued to put a message on a cluster queue. At the time of the call, there are no longer any instances of the queue in the cluster. The MQPUT fails and the message is not sent.

The error can occur if MQOO_BIND_NOT_FIXED is specified on the MQOPEN call that opens the queue, or MQPUT1 is used to put the message.

MQRC_STOPPED_BY_CLUSTER_EXIT (2188 X'88C')

An MQOPEN, MQPUT, or MQPUT1 call is issued to open or put a message on a cluster queue. The cluster workload exit rejects the call.

z/OS Using and writing applications on IBM MQ for z/OS

IBM 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 information explains the IBM MQ facilities available to programs running in each of the supported environments. In addition,

- For information about using the IBM MQ-CICS bridge, see Using IBM MQ with CICS .
- For information about using IMS and the IMS bridge, see “IMS and IMS bridge applications on IBM MQ for z/OS” on page 66.

Use the following links to find out more about using and writing applications on IBM MQ for z/OS:

- “Environment-dependent IBM MQ for z/OS functions” on page 857
- “Debugging facilities, syncpoint support, and recovery support” on page 858
- “The IBM MQ for z/OS interface with the application environment” on page 859
- “Writing z/OS UNIX System Services applications” on page 860
- “Application programming with shared queues” on page 863

Related concepts

“The Message Queue Interface overview” on page 697
Learn about the Message Queue Interface (MQI) components.

“Connecting to and disconnecting from a queue manager” on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

“Opening and closing objects” on page 716
This information provides an insight into opening and closing IBM MQ objects.

“Putting messages on a queue” on page 727
Use this information to learn how to put messages on a queue.

“Getting messages from a queue” on page 741
Use this information to learn about getting messages from a queue.

“Inquiring about and setting object attributes” on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

“Committing and backing out units of work” on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

“Starting IBM MQ applications using triggers” on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

“Working with the MQI and clusters” on page 852
There are special options on calls and return codes that relate to clustering.

“IMS and IMS bridge applications on IBM MQ for z/OS” on page 66
This information helps you to write IMS applications using IBM MQ.

z/OS *Environment-dependent IBM MQ for z/OS functions*

Use this information when considering IBM MQ for z/OS functions.

The main differences to be considered between IBM MQ functions in the environments in which IBM MQ for z/OS runs are:

- IBM 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 IBM 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 Planning on z/OS *IBM MQ for z/OS Concepts and Planning Guide*.

These features are summarized in Table 130 on page 858.

Table 130. 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.

z/OS *Debugging facilities, syncpoint support, and recovery support*

Use this information to learn about program debugging facilities, syncpoint support, and recovery support.

Program debugging facilities

IBM 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 IBM 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 IBM 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 IBM MQ for z/OS.

z/OS batch applications can use IBM 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 Syncpoints in z/OS batch applications.

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.

z/OS *The IBM 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, IBM MQ for z/OS provides an *adapter* for each of the environments it supports.

These adapters are the interface between application programs and IBM MQ for z/OS subsystems. They allow the programs to use the MQI.

z/OS *The batch adapter*

Use this information to learn about the batch adapter and the commit protocol it supports.

The *batch adapter* provides access to IBM 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 IBM MQ for z/OS are at the task level. The adapter provides a single connection thread from an application task control block (TCB) to IBM MQ for z/OS.

The adapter supports a single-phase commit protocol for changes made to resources owned by IBM MQ for z/OS ; it does not support multiphase-commit protocols.

z/OS *The RRS batch adapter*

Use this information to learn about the RRS batch adapter and the two RRS batch adapters provided by IBM MQ.

The transaction management and recoverable resource manager services (RRS) adapter:

- Uses z/OS RRS for commit control.
- Supports simultaneous connections to multiple IBM 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 IBM 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 an IBM MQ batch thread between TCBs.

IBM MQ for z/OS provides two RRS batch adapters:

CSQBRSTB
: This adapter requires you to change any MQCMIT statement to SRRCMIT and any MQBACK statement to SRRBACK in your IBM MQ application. (If you code MQCMIT or MQBACK in an application linked with CSQBRSTB, you receive MQRC_ENVIRONMENT_ERROR.)

CSQBRRSI
This adapter allows your IBM 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 IBM MQ applications to use RRS coordination with few or no changes.

If you link-edit your IBM MQ application with the CSQBRRSI adapter, MQCMIT and MQBACK syncpoint your unit of work across IBM MQ and all other RRS-enabled resource managers. If you link-edit your IBM MQ application with the CSQBRSTB adapter, change MQCMIT to SRRCMIT and MQBACK to SRRBACK. The latter approach is preferable; it clearly indicates that the syncpoint is not restricted to IBM MQ resources only.

z/OS *The IMS adapter*
If you are using the IMS adapter from an IBM 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 IBM 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 IBM MQ. The adapter supports a two-phase commit protocol for changes made to resources owned by IBM 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 IBM MQ applications using triggers” on page 834.

If you are writing batch DL/I programs, follow the guidance given in this topic for z/OS batch programs.

z/OS *Writing z/OS UNIX System Services applications*

The batch adapter supports queue manager connections from batch and TSO address spaces.

For 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 UNIX System Services 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.)

IBM MQ for z/OS supports z/OS UNIX System Services 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, IBM 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 IBM MQ resources from multiple threads.

z/OS *The API-crossing exit for z/OS*

This topic contains product-sensitive programming interface information.

An exit is a point in IBM-supplied code where you can run your own code. IBM 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 IBM MQ for z/OS.

This section is applicable only for users of CICS TS V3.1 and earlier. Users of CICS TS V3.2 and later should refer to the section CICS Integration with IBM MQ in the CICS product documentation.

Note

The API-crossing exit is invoked only by the CICS adapter of IBM MQ for z/OS. The exit program runs in the CICS address space.

z/OS *Writing your own exit program*

You can use the sample API-crossing exit program (CSQCAPX) that is supplied with IBM MQ for z/OS as a framework for your own program.

This is described in "The sample API-crossing exit program, CSQCAPX" on page 862.

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 IBM 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 z/OS 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 IBM MQ for z/OS code. Ensure that your exit does not disrupt any IBM MQ for z/OS programs or transactions that use the MQI. These are typically 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 IBM 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).

z/OS *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 correct 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, IBM MQ for z/OS performance is degraded.

z/OS *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 "Building CICS applications in z/OS" on page 988.

- For CICS Transaction Server for z/OS 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 IBM MQ for z/OS with the API-crossing exit permanently enabled, with CICS Transaction Server for z/OS 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.

z/OS *Application programming with shared queues*

This topic provides information on 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.

z/OS *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 IBM 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.

IBM 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. IBM 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 IBM 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: IBM MQ only guarantees to preserve the sequence of messages when certain criteria are met. These are described in the description of MQGET.

z/OS *Applications that are not suitable for use with shared queues*

Some features of IBM 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.

z/OS *Deciding whether to share non-application queues*

Use this information when considering sharing 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 131 on page 866.

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 IBM 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. See Defining system objects for IBM MQ for z/OS for more information.

You cannot define any other SYSTEM.* queues as shared queues.

z/OS *Migrating your existing applications to use shared queues*

Reason codes, triggering, and the MQINQ API call can work differently in a shared queue environment.

See Migrating non-shared queues to shared queues for information on migrating your existing queues to shared queues.

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 131 on page 866 describes what you need to consider when deciding which type of initiation queue to use.

Table 131. 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 particular 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.	For trigger type EVERY, when an application puts a message to a shared application queue, the putting queue manager determines which queue managers have an interest in the trigger-every event and sends a notification to one of those queue managers. On the notified queue manager, the resulting action is to generate a trigger message to the initiation queue., **Note:** If you have a shared application queue that has a trigger type of EVERY, use a shared initiation queue, or you might lose trigger messages in certain circumstances; for example, a queue manager failing. For trigger type FIRST or DEPTH, one trigger message is generated by each queue manager that has the named initiation queue open for input. **Note:** For trigger type FIRST or DEPTH, if one trigger monitor instance is busy, this leaves the potential for less busy trigger monitors to process more than one trigger message from the shared initiation queue. Hence, multiple instances of the server application may be started against a given queue manager. Note that these multiple instances are started as a result of processing multiple trigger messages. Ordinarily, for trigger type FIRST or DEPTH, if an application instance is already serving an application queue, another trigger message will not be generated by the queue manager that the application is connected to.

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.

z/OS IMS and IMS bridge applications on IBM MQ for z/OS

This information helps you to write IMS applications using IBM MQ.

- To use syncpoints and MQI calls in IMS applications, see "Writing IMS applications using IBM MQ" on page 67.
- To write applications that use the IBM MQ - IMS bridge, see "Writing IMS bridge applications" on page 71.

Use the following links to find out more about IMS and IMS bridge applications on IBM MQ for z/OS:

- "Writing IMS applications using IBM MQ" on page 67
- "Writing IMS bridge applications" on page 71

Related concepts

"The Message Queue Interface overview" on page 697
Learn about the Message Queue Interface (MQI) components.

"Connecting to and disconnecting from a queue manager" on page 709
To use IBM MQ programming services, a program must have a connection to a queue manager. Use this information to learn how to connect to and disconnect from a queue manager.

"Opening and closing objects" on page 716
This information provides an insight into opening and closing IBM MQ objects.

"Putting messages on a queue" on page 727
Use this information to learn how to put messages on a queue.

"Getting messages from a queue" on page 741
Use this information to learn about getting messages from a queue.

"Inquiring about and setting object attributes" on page 820
Attributes are the properties that define the characteristics of an IBM MQ object.

"Committing and backing out units of work" on page 823
This information describes how to commit and back out any recoverable get and put operations that have occurred in a unit of work.

"Starting IBM MQ applications using triggers" on page 834
Learn about triggers and how to start IBM MQ applications using triggers.

"Working with the MQI and clusters" on page 852
There are special options on calls and return codes that relate to clustering.

"Using and writing applications on IBM MQ for z/OS" on page 857
IBM 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.

z/OS *Writing IMS applications using IBM MQ*

There are further considerations when using IBM MQ in IMS applications These include which MQ API calls can be used and the mechanism used for syncpoint.

Use the following links to find out more about writing IMS applications on IBM MQ for z/OS:

- "Syncpoints in IMS applications" on page 67
- "MQI calls in IMS applications" on page 68

Restrictions

There are restrictions on which IBM MQ API calls can used by an application using the IMS adapter.

The following IBM MQ API calls are not supported within an application using the IMS adapter:

- MQCB
- MQCB_FUNCTION

- MQCTL

Related concepts

“Writing IMS bridge applications” on page 71
This topic contains information about writing applications to use the IBM MQ - IMS bridge.

z/OS *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 ROLB call in the IMS documentation.

The queue manager is a participant in a two-phase commit protocol; the IMS syncpoint manager is the coordinator.

All open handles are closed by the IMS adapter at a syncpoint (except in a batch or non-message driven BMP environment). This is because a different user could initiate the next unit of work and IBM MQ security checking is performed when the MQCONN, MQCONNX, and MQOPEN calls are made, not when the MQPUT or MQGET calls are made.

However, in a Wait-for-Input (WFI) or pseudo Wait-for-Input (PWFI) environment IMS does not notify IBM MQ to close the handles until either the next message arrives or a QC status code is returned to the application. If the application is waiting in the IMS region and any of these handles belong to triggered queues, triggering will not occur because the queues are open. For this reason, applications running in a WFI or PWFI environment should explicitly MQCLOSE the queue handles before doing the GU to the IOPCB for the next message.

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.

z/OS *MQI calls in IMS applications*

Use this information to learn about the use of MQI calls on Server applications and Enquiry applications.

This section covers the use of MQI calls in the following types of IMS applications:

- “Server applications” on page 868
- “Inquiry applications” on page 870

Server applications

Here is an outline of the MQI server application model:

```
Initialize/Connect
.
Open queue for input shared
.
Get message from IBM 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 IBM 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 IBM MQ:

```
main()
----
Call InitIMS
If IMS initialization successful
Call InitMQM
If IBM 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 IBM 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 IBM 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 IBM MQ IMS trigger monitor program (CSQQTRMN) to schedule the MPP transaction when there are messages on the IBM 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.

Inquiry applications

A typical IBM MQ application initiating an inquiry or update works as follows:

- Gather data from the user

- Put one or more IBM MQ messages
- Get the reply messages (you might have to wait for them)
- Provide a response to the user

Because messages put on to IBM MQ queues do not become available to other IBM 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 Intersystem Communication (ISC)* 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 IBM MQ queue
.
Switch to second IBM 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 IBM MQ queue
.
Return results to originator
.
END
```

z/OS *Writing IMS bridge applications*

This topic contains information about writing applications to use the IBM MQ - IMS bridge.

For information about the IBM MQ - IMS bridge, see The IMS bridge.

Use the following links to find out more about writing IMS bridge applications on IBM MQ for z/OS:

- “How the IMS bridge deals with messages” on page 71
- “Writing IMS transaction programs through IBM MQ” on page 878

Related concepts

“Writing IMS applications using IBM MQ” on page 67
There are further considerations when using IBM MQ in IMS applications These include which MQ API calls can be used and the mechanism used for syncpoint.

z/OS *How the IMS bridge deals with messages*

When you use the IBM 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 IBM 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.

The IBM MQ-IMS bridge requires exclusive input access (MQOO_INPUT_EXCLUSIVE) to the bridge queue if it is defined with QSGDISP(QMGR), or if it is defined with QSGDISP(SHARED) together with the NOSHARE option.

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 IBM MQ connects to IMS, and is described in Application access control for the IMS bridge. This enables a pseudo signon to be implemented.

The IBM MQ - IMS bridge accepts the following types of message:

- Messages containing IMS transaction data and an MQIIH structure (described in MQIIH):

```
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>]
```

IBM 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 IBM 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 MQIIH), 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 IBM MQ - IMS bridge uses one or two Tpipes for each IBM 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 IBM 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.

See the following topics for more information about how the IBM MQ - IMS bridge deals with messages:

- “Mapping IBM MQ messages to IMS transaction types” on page 73
- “If the message cannot be put to the IMS queue” on page 73
- “IMS bridge feedback codes” on page 74
- “The MQMD fields in messages from the IMS bridge” on page 74
- “The MQIIH fields in messages from the IMS bridge” on page 75
- “Reply messages from IMS” on page 76
- “Using alternate response PCBs in IMS transactions” on page 76

- “Sending unsolicited messages from IMS” on page 76
- “Message segmentation” on page 77
- “Data conversion for messages to and from the IMS bridge” on page 77

Related concepts

“Writing IMS transaction programs through IBM MQ” on page 878
The coding required to handle IMS transactions through IBM 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.

z/OS *Mapping IBM MQ messages to IMS transaction types*
A table describing the mapping of IBM MQ messages to IMS transaction types.

Table 132. How IBM MQ messages map to IMS transaction types

IBM MQ message type	**Commit-then-send (mode 0) - uses synchronized IMS Tpipes**	**Send-then-commit (mode 1) - uses non-synchronized IMS Tpipes**
Persistent IBM MQ messages	• Recoverable full function transactions • Unrecoverable transactions are rejected by IMS	• Fastpath transactions • Conversational transactions • Full function transactions
Nonpersistent IBM MQ messages	• Unrecoverable full function transactions • Recoverable transactions are permitted with IMS V8 and APAR PQ61404 and all later versions of IMS	• Fastpath transactions • Conversational transactions • Full function transactions

Note: IMS commands cannot use persistent IBM MQ messages with commit mode 0. See Commit mode (commitMode) for more information.

z/OS *If the message cannot be put to the IMS queue*
Learn about actions to take 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 IBM 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, IBM MQ sends an error message to the system console, the message includes the IMS sense code, and the IBM MQ message is put to the dead-letter queue. If the IMS sense code is 001A, IMS sends an IBM MQ message containing the reason for the failure to the reply-to queue.

  Note: In the circumstances listed previously, if IBM MQ cannot put the message to the dead-letter queue for any reason, the message is returned to the originating IBM 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 IBM MQ subsystem

- If the message is rejected by IMS for anything other than a message error, the IBM MQ message is returned to the originating queue, IBM 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.

z/OS *IMS bridge feedback codes*

IMS sense codes are typically output in hexadecimal format in IBM MQ console messages such as CSQ2001I (for example, sense code 0x001F). IBM 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, or 600 through 855 for NACK sense code 0x001A. 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 IBM MQ *Feedback* code.
3. The IMS-OTMA sense code 0x001A, decimal 26 is a special case. A *Feedback* code in the range 600-855 is generated.
   a. The IMS-OTMA reason code is converted from a hexadecimal number to a decimal number.
   b. 600 is added to the number resulting from the calculation in a, giving the IBM MQ *Feedback* code.

For information about IMS-OTMA sense codes, see OTMA sense codes for NAK messages.

z/OS *The MQMD fields in messages from the IMS bridge*

Learn about 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

z/OS *The MQIIH fields in messages from the IMS bridge*

Learn about 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. In versions of IMS prior to V14, this field is always nulls if not in conversation. From IMS V14 onwards, this field may be set by IMS even if not in conversation.

TranState
"C" if in conversation, otherwise blank

CommitMode
Commit mode from OTMA header ("0" or "1")

SecurityScope
Blank

Reserved
Blank

z/OS *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 IBM 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 acknowledgment 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.

z/OS *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 Using OTMA exits in IMS and The pre-routing exit DFSYPRX0 for information about these exit programs.

Unless action is taken in the exits, all output from IMS transactions initiated from an IBM MQ queue manager, whether to the IOPCB or to an ALTPCB, will be returned to the same queue manager.

z/OS *Sending unsolicited messages from IMS*

To send messages from IMS to an IBM 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 pre-routing exit DFSYPRX0 and The destination resolution user exit for information about these exit programs.

Note: The IBM 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.

z/OS *Message segmentation*

You can define IMS transactions as expecting single- or multi-segment input.

The originating IBM 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 IBM MQ message sent with a single MQPUT.

The maximum length of an LLZZ-data segment is defined by IMS/OTMA (32767 bytes). The total IBM 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 IBM 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.

z/OS *Data conversion for messages to and from the IMS bridge*

The data conversion is performed by either the distributed queuing facility (which may call any necessary exits) or by the intra group queuing agent (which does not support the use of exits) when it puts a message to a destination queue that has XCF information defined for its storage class. The data conversion does not occur when a message is delivered to a queue by publish/subscribe.

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 IBM MQ - IMS bridge from any IBM MQ platform.

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 IBM 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 947 for detailed information about data conversion in general.

Sending messages to the IBM 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 can 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 IBM 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.

z/OS *Writing IMS transaction programs through IBM MQ*

The coding required to handle IMS transactions through IBM 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 in the range 1 through 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.

Consider the following topics when writing code to handle IMS transactions through IBM MQ.

- “Writing IBM MQ applications to invoke IMS conversational transactions” on page 879
- “Writing programs containing IMS commands” on page 879

- "Triggering" on page 879

Writing IBM MQ applications to invoke IMS conversational transactions

Use this information as a guide for considerations when writing IBM MQ application 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 IBM 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 IBM MQ - IMS bridge are terminated if IMS is restarted.

Writing programs containing IMS commands

An application program can build an IBM 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 *IMS V11 Communications and Connections* for details. The command output is received in the IBM 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 *IMS V11 Communications and Connections*. To invoke this form from an IBM 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.

Triggering

The IBM 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 client procedural applications

What you need to know to write client applications on IBM MQ using a procedural language.

Applications can be built and run in the IBM MQ client environment. The application must be built and linked to the IBM MQ MQI client used. The way in which applications are built and linked varies according to the platform and programming language used. For information on how to build client applications, see "Building applications for IBM MQ MQI clients" on page 885.

You can run an IBM MQ application both in a full IBM MQ environment and in an IBM MQ MQI client environment without changing your code, provided that certain conditions are met. For more information on running your applications in the IBM MQ client environment, see “Running applications in the IBM MQ MQI client environment” on page 887.

If you use the message queue interface (MQI) to write applications to run in an IBM MQ MQI client environment there are some additional controls to impose during an MQI call to ensure that the IBM MQ application processing is not disrupted. For more information about these controls, see “Using the MQI in a client application” on page 880.

See the following topics for information preparing and running other application types as client applications:

- “Preparing and running CICS and Tuxedo applications” on page 899
- “Preparing and running Microsoft Transaction Server applications” on page 47
- “Preparing and running IBM MQ JMS applications” on page 902

Related concepts

“Application development concepts” on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

“Developing applications for IBM MQ” on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

“Design considerations for IBM MQ applications” on page 47
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 IBM MQ.

“Writing a procedural application for queuing” on page 696
Use this information to learn about writing queuing applications, connecting to and disconnecting from a queue manager, publish/subscribe, and opening and closing objects.

“Writing publish/subscribe applications” on page 780
Start writing publish/subscribe IBM MQ applications.

“Building a procedural application” on page 962
You can write an IBM MQ application in one of several procedural languages, and run the application on several different platforms.

“Handling procedural program errors” on page 999
This information explains errors associated with your applications MQI calls either when it makes a call, or when its message is delivered to its final destination.

Related tasks

“Using the IBM MQ sample procedural programs” on page 1018
These sample programs are written in procedural languages, and demonstrate typical uses of the Message Queue Interface (MQI). IBM MQ programs on different platforms.

Using the MQI in a client application

This collection of topics considers the differences between writing your IBM MQ application to run in a message queue interface (MQI) client environment and to run in the full IBM MQ queue manager environment.

When you design an application, consider what controls you need to impose during an MQI call to ensure that the IBM MQ application processing is not disrupted.

Before you can run applications that use the MQI you must create certain IBM MQ objects. For more information, see Application programs using the MQI.

Limiting the size of a message in a client application

A queue manager has a maximum message length, but the maximum size of message you can transmit from a client application is limited by the channel definition.

The maximum message length (MaxMsgLength) attribute of a queue manager is the maximum length of a message that can be handled by that queue manager.

Multi On Multiplatforms, you can increase the maximum message length attribute of a queue manager. For more information, see ALTER QMGR.

You can find out the value of MaxMsgLength for a queue manager by using the MQINQ call.

If the MaxMsgLength attribute is changed, no check is made that there are not already queues, and even messages, with a length greater than the new value. After you change this attribute, restart applications and channels in order to ensure that the change has taken effect. It is then not possible for any new messages to be generated that exceed the MaxMsgLength of either the queue manager or the queue (unless queue manager segmentation is allowed).

The maximum message length in a channel definition limits the size of a message that you can transmit along a client connection. If an IBM MQ application tries to use the MQPUT call or the MQGET call with a message larger than this, an error code is returned to the application. The maximum message size parameter of the channel definition does not affect the maximum message size which can be consumed using MQCB over a client connection.

Related concepts

"Using MQCONNX" on page 885
You can use the MQCONNX call to specify a channel definition (MQCD) structure in the MQCNO structure.

Related reference

Maximum message length (MAXMSGL)
ALTER CHANNEL
2010 (07DA) (RC2010): MQRC_DATA_LENGTH_ERROR

Choosing client or server CCSID

Use the local coded character set identifier (CCSID) for the client. The queue manager performs necessary conversion. You can use the **MQCCSID** environment variable to override the CCSID. If your application performs multiple PUTs, the CCSID and encoding fields of the MQMD can be overwritten after completion of the first PUT.

The data passed across the message queue interface (MQI) from the application to the client stub must be in the local CCSID, encoded for the IBM MQ MQI client. If the connected queue manager requires the data to be converted, then conversion is done by the client support code on the queue manager.

In IBM WebSphere MQ 7.0 and later versions, the Java client can do the conversion if the queue manager is unable to do so. See "IBM MQ classes for Java client connections" on page 359.

The client code assumes that the character data crossing the MQI in the client is in the CCSID configured for that workstation. If this CCSID is an unsupported CCSID or is not the required CCSID, it can be overridden with the **MQCCSID** environment variable by using one of these commands:

- Windows

```
SET MQCCSID=850
```

- Linux AIX

```
export MQCCSID=850
```

- IBM i

```
ADDENVVAR ENVVAR(MQCCSID) VALUE(37)
```

If this parameter is set in the profile, all MQI data is assumed to be in code page 850.

Note: The assumption about code page 850 does not apply to application data in the message.

If your application is performing multiple PUTs that include IBM MQ headers after the message descriptor (MQMD), be aware that the CCSID and encoding fields of the MQMD are overwritten after completion of the first PUT.

After the first PUT, these fields contain the value used by the connected queue manager to convert the IBM MQ headers. Ensure that your application resets the values to the values it requires.

Using MQINQ in a client aplication

Some values queried using MQINQ are modified by the client code.

CCSID
is set to the client CCSID, not that of the queue manager.

MaxMsgLength
is reduced if it is restricted by the channel definition. This will be the lower of:

- The value defined in the queue definition, or
- The value defined in the channel definition

For more information, see the MQINQ.

Using sync point coordination in a client application

An application running on the base client can issue MQCMIT and MQBACK, but the scope of the sync point control is limited to the MQI resources. You can use an external transaction manager with an extended transactional client.

Within IBM MQ, one of the roles of the queue manager is sync point control within an application. If an application runs on an IBM MQ base client, it can issue MQCMIT and MQBACK, but the scope of the sync point control is limited to the MQI resources. The IBM MQ verb MQBEGIN is not valid in a base client environment.

Applications running in the full queue manager environment on the server can coordinate multiple resources (for example databases) via a transaction monitor. On the server you can use the Transaction Monitor supplied with IBM MQ products, or another transaction monitor such as CICS. You cannot use a transaction monitor with a base client application.

You can use an external transaction manager with an IBM MQ extended transactional client. See What is an extended transactional client? for details.

Using read ahead in a client application

You can use read ahead on a client to allow non persistent messages to be sent to a client without the client application having to request the 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.

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 benefit from the performance improvements when consuming non persistent messages.

When you call MQOPEN with MQOO_READ_AHEAD, the IBM MQ client only enables read-ahead if certain conditions are met. These conditions include:

- The client application must be compiled and linked against the threaded IBM MQ MQI client libraries.
- The client channel must be using TCP/IP protocol
- The channel must have a non-zero SharingConversations (SHARECNV) setting in both the client and server channel definitions.

When read ahead is enabled, messages are sent to a memory buffer on the client called the read ahead buffer. The client has 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.

Not all client application designs are suited to using read ahead because not all options are supported for use. 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 remain stranded in the client read ahead buffer. For more information, see "Improving performance of non-persistent messages" on page 759

Read ahead configuration is controlled by three attributes, MaximumSize, PurgeTime, and UpdatePercentage, which are specified in the MessageBuffer stanza of the IBM MQ client configuration file.

Using asynchronous put in a client application

Using asynchronous put, an application can put a message to a queue without waiting for a response from the queue manager. You can use this to improve messaging performance in some situations.

Normally, when an application puts a message or messages on a queue, using MQPUT or MQPUT1, the application has to wait for the queue manager to confirm that it has processed the MQI request. You can improve messaging performance, particularly for applications that use client bindings, and applications that put large numbers of small messages to a queue, by choosing instead to put messages asynchronously. When an application puts a message asynchronously, the queue manager does not return the success or failure of each call, but you can instead check for errors periodically.

To put a message on a queue asynchronously, use the MQPMO_ASYNC_RESPONSE option in the *Options* field of the MQPMO structure.

If a message is not eligible for asynchronous put, it is put to a queue synchronously.

When requesting asynchronous put response for MQPUT or MQPUT1, a CompCode and Reason of MQCC_OK and MQRC_NONE does not necessarily mean that the message was successfully put to a queue. Although the success or failure of each individual MQPUT or MQPUT1 call might not be returned immediately, the first error that occurred under an asynchronous call can be determined later through a call to MQSTAT.

For more details on MQPMO_ASYNC_RESPONSE, see MQPMO options.

The Asynchronous Put sample program demonstrates some of the features available. For details of the features and design of the program, and how to run it, see "The Asynchronous Put sample program" on page 1037.

Using sharing conversations in a client application

In an environment where sharing conversations is permitted, conversations can share an MQI channel instance.

Sharing conversations is controlled by two fields, both called SharingConversations, one of which is part of the channel definition (MQCD) structure and one of which is part of the channel exit parameter (MQCXP) structure. The SharingConversations field in the MQCD is an integer value, determining the maximum number of conversations that can share a channel instance associated with the channel. The SharingConversations field in the MQCXP is a boolean value, indicating whether the channel instance is currently shared.

In an environment where sharing conversations is not permitted, new client connections specifying identical MQCDs will not share a channel instance.

A new client application connection will share the channel instance when the following conditions are true:

- Both the client-connection and server-connection ends of the channel instance are configured for sharing conversations, and these values are not overridden by channel exits.
- The client connection MQCD value (supplied on the client MQCONNX call or from the client channel definition table (CCDT)) exactly matches the client connection MQCD value supplied on the client MQCONNX call or from the CCDT when the existing channel instance was first established. Note that the original MQCD might have been subsequently altered by exits or by channel negotiation, but that the match is made against the value which was supplied to the client system before these changes were made.
- The sharing conversations limit on the server side is not exceeded.

If a new client application connection matches the criteria to run sharing a channel instance with other conversations, this decision is made before any exits are called on that conversation. Exits on such a conversation cannot alter the fact that it is sharing the channel instance with other conversations. If there are no existing channel instances matching the new channel definition, a new channel instance is connected.

Channel negotiation only occurs for the first conversation on a channel instance; the negotiated values for the channel instance are fixed at that stage and cannot be altered when subsequent conversations start. TLS authentication also only occurs for the first conversation.

If the MQCD SharingConversations value is altered during the initialization of any security, send or receive exits for the first conversation on the socket at either the client-connection or the server-connection end of the channel instance, the new value it has after all these exits are initialized is used to determine the sharing conversations value for the channel instance (the lowest value takes precedence).

If the negotiated value for sharing conversations is zero, the channel instance is never shared. Further exit programs that set this field to zero similarly run on their own channel instance.

If the negotiated value for sharing conversations is greater than zero then MQCXP SharingConversations is set to TRUE for subsequent calls to exits, indicating that other exit programs on this channel instance can be entered simultaneously with this one.

When you write a channel exit program, consider whether it will run on a channel instance that might involve sharing conversations. If the channel instance might involve sharing conversations, consider the effect on other instances of the channel exit of changing MQCD fields; all MQCD fields have common values across all the sharing conversations. After the channel instance is established, if exit programs try to alter MQCD fields they might encounter problems because other instances of exit programs running on the channel instance could be attempting to alter the same fields at the same time. If this situation could arise with your exit programs, you must serialize access to the MQCD in your exit code.

If you are working with a channel which is defined to share conversations, but you do not want sharing to occur on a particular channel instance, set the MQCD value of SharingConversations to 1 or 0 when you initialize a channel exit on the first conversation on the channel instance. See SharingConversations for an explanation of the values of SharingConversations.

Example

Sharing conversations is enabled.

You are using a client-connection channel definition which specifies an exit program.

The first time that this channel starts, the exit program alters some of the MQCD parameters when it is initialized. These are acted on by the channel, so the definition that the channel is running with is now different from the one that was originally supplied. The MQCXP SharingConversations parameter is set to TRUE.

The next time that the application connects using this channel, the conversation runs on the channel instance which was started previously, because it has the same original channel definition. The channel instance the application connects to the second time is the same instance as the first time it connected. Consequently, it uses the definitions that have been altered by the exit program. When the exit program

is initialized for the second conversation, although it can alter MQCD fields, they are not acted on by the channel. These same characteristics apply to any subsequent conversations which share the channel instance.

Using MQCONNX

You can use the MQCONNX call to specify a channel definition (MQCD) structure in the MQCNO structure.

This allows the calling client application to specify the definition of the client-connection channel at run time. For more information, see Creating a client-connection channel on the IBM MQ MQI client using MQCNO. When you use MQCONNX, the call issued at the server depends on the server level and listener configuration.

When you use MQCONNX from a client, the following options are ignored:

- MQCNO_STANDARD_BINDING
- MQCNO_FASTPATH_BINDING

The MQCD structure you can use depends on the MQCD version number you are using. For information on MQCD versions (MQCD_VERSION), see MQCD Version. You can use the MQCD structure, for example, to pass channel-exit programs to the server. If you are using MQCD Version 3 or later, you can use the structure to pass an array of exits to the server. You can use this function to perform more than one operation on the same message, such as encryption and compression, by adding an exit for each operation, rather than modifying an existing exit. If you do not specify an array in the MQCD structure, the single exit fields will be checked. For more information on channel-exit programs, see “Channel-exit programs for messaging channels” on page 926.

Shared connection handles on MQCONNX

You can share handles between different threads within the same process, using shared connection handles.

When you specify a shared connection handle, the connection handle returned from the MQCONNX call can be passed in subsequent MQI calls on any thread in the process.

Note: You can use a shared connection handle on an IBM MQ MQI client to connect to a server queue manager that does not support shared connection handles.

Building applications for IBM MQ MQI clients

Applications can be built and run in IBM MQ MQI client environment. The application must be built and linked to the IBM MQ MQI client used. The way in which applications are built and linked varies according to the platform and programming language used.

If an application is to run in a client environment, you can write it in the languages shown in the following table:

Table 133. Programming languages supported in client environments

Client platform	C	C++	COBOL	pTAL	RPG	Visual Basic
AIX	Yes	Yes	Yes			
IBM i	Yes		Yes		Yes	
Linux	Yes	Yes	Yes			
Windows	Yes	Yes	Yes			Yes

Multi *Linking C applications with the IBM MQ MQI client code*

Having written your IBM MQ application that you want to run on the IBM MQ MQI client, you must link it to the IBM MQ MQI client code.

You can link your application to the IBM MQ MQI client code in two ways:

1. Directly, by connecting your application to a queue manager, in which case the queue manager must be on the same machine as your application.
2. To a client library file, which gives you access to queue managers on the same or on a different machine.

IBM MQ provides a client library file for each environment:

AIX **AIX**
libmqic.a library for non-threaded applications, or libmqic_r.a library for threaded applications.

Linux **Linux**
libmqic.so library for non-threaded applications, or libmqic_r.so library for threaded applications.

IBM i **IBM i**
Bind client application with LIBMQIC client service program for non-threaded applications, or LIBMQIC_R service program for threaded applications.

Windows **Windows**
MQIC32.LIB.

ALW *Linking C++ applications with the IBM MQ MQI client code*

You can write applications to run on the client in C++. Build methods vary according to the environment.

For information about how to link your C++ applications, see Building IBM MQ C++ programs.

For full details of all aspects of using C++, see Using C++

Multi *Linking COBOL applications with the IBM MQ MQI client code*

Having written a COBOL application that you want to run on the IBM MQ MQI client, you must link it with an appropriate library.

IBM MQ provides a client library file for each environment:

AIX **AIX**
Link your non-threaded COBOL application with the library libmqicb.a or threaded COBOL application with libmqicb_r.a.

IBM i **IBM i**
Bind COBOL client application with AMQCSTUB service program for non-threaded applications, or AMQCSTUB_R service program for threaded applications.

Windows **Windows**
Link your application code with the MQICCBB library for 32-bit COBOL. The IBM MQ MQI client for Windows does not support 16-bit COBOL.

Windows *Linking Visual Basic applications with the IBM MQ MQI client code*

You can link Microsoft Visual Basic applications with the IBM MQ MQI client code on Windows.

Deprecated

From IBM MQ 9.0, support for Microsoft Visual Basic 6.0 is deprecated. IBM MQ classes for .NET are the recommended replacement technology. For more information, see Developing .NET applications.

Link your Visual Basic application with the following include files:

CMQB.bas
MQI

CMQBB.bas
MQAI

CMQCFB.bas
PCF commands

CMQXB.bas
Channels

Set `mqtype=2` for the client in the Visual Basic compiler, to ensure the correct automatic selection of the client dll:

MQIC32.dll
Windows 7, Windows 8, Windows 2008, and Windows 2012

Related concepts

"Coding in Visual Basic" on page 1012
Information to consider when coding IBM MQ programs in Microsoft Visual Basic. Visual Basic is supported only on Windows.

"Preparing Visual Basic programs in Windows" on page 981
Information to consider when using Microsoft Visual Basic programs on Windows.

Running applications in the IBM MQ MQI client environment

You can run an IBM MQ application both in a full IBM MQ environment and in an IBM MQ MQI client environment without changing your code, provided that certain conditions are met.

These conditions are that:

- The application does not need to connect to more than one queue manager concurrently.
- The queue manager name is not prefixed with an asterisk (*) on an `MQCONN` or `MQCONNX` call.
- The application does not need to use any of the exceptions listed in What applications run on an IBM MQ MQI client?

Note: The libraries that you use at link-edit time determine the environment in which your application must run.

When working in the IBM MQ MQI client environment, remember that:

- Each application running in the IBM MQ MQI client environment has its own connections to servers. An application establishes one connection to a server each time it issues an `MQCONN` or `MQCONNX` call.
- An application sends and gets messages synchronously. This implies a wait between the time the call is issued at the client and the return of a completion code and reason code across the network.
- All data conversion is done by the server, but see also MQCCSID for information about overriding the machine's configured CCSID.

Connecting IBM MQ MQI client applications to queue managers

An application running in an IBM MQ MQI client environment can connect to a queue manager in various ways. You can use environment variables, the MQCNO structure, or a client definition table.

When an application running in an IBM MQ client environment issues an MQCONN or MQCONNX call, the client identifies how it is to make the connection. When an MQCONNX call is issued by an application on an IBM MQ client, the MQI client library searches for the client channel information in the following order:

1. Using the contents of the `ClientConnOffset` or `ClientConnPtr` fields of the MQCNO structure (if supplied). These fields identify the channel definition structure (MQCD) to be used as the definition of the client connection channel. Connection details can be overridden by using a pre-connect exit. For more information, see "Referencing connection definitions using a pre-connect exit from a repository" on page 956.
2. If the **MQSERVER** environment variable is set, the channel that it defines is used.

3. If an `mqclient.ini` file is defined and the Channels stanza contains a **ServerConnectionParms** attribute, the channel that it defines is used. For more information, see IBM MQ MQI client configuration file, `mqclient.ini` and Channels stanza of the client configuration file.
4. If the **MQCHLLIB** and **MQCHLTAB** environment variables are set, the client channel definition table that they point to is used. Alternatively, the **MQCCDTURL** environment variable provides the equivalent capability to setting a combination of the **MQCHLLIB** and **MQCHLTAB** environment variables. If **MQCCDTURL** is set, the client channel definition table that it points to is used. For more information, see URL access to the CCDT.
5. If an `mqclient.ini` file is defined and the Channels stanza contains **ChannelDefinitionDirectory** and **ChannelDefinitionFile** attributes, these attributes are used to locate the client channel definition table. For more information, see IBM MQ MQI client configuration file, `mqclient.ini` and Channels stanza of the client configuration file.
6. Finally, if the environment variables are not set, the client searches for a client channel definition table with a path and name that are established from the **DefaultPrefix** attribute of the AllQueueManagers stanza in the `mqs.ini` file. For more information, see AllQueueManagers stanza of the mqs.ini file.

   If the search for a client channel definition table fails, the client uses the following paths:

   - Linux AIX On AIX and Linux: `/var/mqm/AMQCLCHL.TAB`
   - Windows On Windows: `C:\Program Files\IBM\MQ\amqclchl.tab`
   - IBM i On IBM i: `/QIBM/UserData/mqm/@ipcc`
   - MQ Appliance On IBM MQ Appliance: *QMname*`_AMQCLCHL.TAB`. They appear under the `mqbackup:// URI`.

The first of the options described in the previous list (using the `ClientConnOffset` or `ClientConnPtr` fields of MQCNO) is supported only by the MQCONNX call. If the application is using MQCONN rather than MQCONNX, the channel information is searched for in the remaining five ways in the order shown in the list. If the client fails to find the channel information, the MQCONN or MQCONNX call fails.

The channel name (for the client connection) must match the server-connection channel name defined on the server for the MQCONN or MQCONNX call to succeed.

Related concepts

Web addressable access to the client channel definition table

Related tasks

Configuring connections between the server and client

Related reference

Client channel definition table

MQCNO - Connect options

Connecting client applications to queue managers using environment variables

Client channel information can be supplied to an application running in a client environment by environment variables.

An application running in an IBM MQ MQI client environment can connect to a queue manager by using the following environment variables:

MQSERVER

The **MQSERVER** environment variable is used to define a minimal channel. **MQSERVER** specifies the location of the IBM MQ server and the communication method to be used.

MQCHLLIB

The **MQCHLLIB** environment variable specifies the directory path to the file containing the client channel definition table (CCDT). The file is created on the server, but can be copied across to the IBM MQ MQI client workstation.

MQCHLTAB
The **MQCHLTAB** environment variable specifies the name of the file containing the client channel definition table (CCDT).

The **MQCCDTURL** environment variable provides the equivalent capability to setting a combination of the **MQCHLLIB** and **MQCHLTAB** environment variables. **MQCCDTURL** allows you to provide a file, ftp, or http URL as a single value from which a client channel definition table can be obtained. For more information, see Web addressable access to the client channel definition table.

Connecting client applications to queue managers using the MQCNO structure
You can specify the definition of the channel in a channel definition structure (MQCD), which is supplied using the MQCNO structure of the MQCONNX call.

For more information, see Creating a client-connection channel on the IBM MQ MQI client using MQCNO.

Connecting client applications to queue managers using a client channel definition table
If you use the MQSC DEFINE CHANNEL command, the details you provide are placed in the client channel definition table (ccdt). The contents of the **QMgrName** parameter of the MQCONN or MQCONNX call determines which queue manager the client connects to.

This file is accessed by the client to determine the channel an application will use. Where there is more than one suitable channel definition, the choice of channel is influenced by the client channel weight (CLNTWGHT) and connection affinity (AFFINITY) channel attributes.

Using automatic client reconnection
You can make your client applications reconnect automatically, without writing any additional code, by configuring a number of components.

Automatic client reconnection is *inline*. The connection is automatically restored at any point in the client application program, and the handles to open objects are all restored.

In contrast, manual reconnection requires the client application to re-create a connection using MQCONN or MQCONNX, and to reopen objects. Automatic client reconnection is suitable for many, but not all client applications.

For more information, see Automatic client reconnection.

Role of the client channel definition table
The client channel definition table (CCDT) contains definitions of client connection channels. It is particularly useful if your client applications might need to connect to a number of alternative queue managers.

The client channel definition table is created when you define a queue manager. The same file can be used by more than one IBM MQ client.

There are a number of ways for a client application to use a CCDT. The CCDT can be copied to the client computer. You can copy the CCDT to a location shared by more than one client. You can make the CCDT accessible to the client as a shared file, while it remains located on the server.

The CCDT can be hosted in a central location that is accessible through a URI, removing the need to individually update the CCDT for each deployed client.

Related concepts
Web addressable access to the client channel definition table
Related tasks
Accessing client-connection channel definitions
Related reference
Client channel definition table

Queue manager groups in the CCDT

You can define a set of connections in the client channel definition table (CCDT) as a *queue manager group*. You can connect an application to a queue manager that is part of a queue manager group. This can be done by prefixing the queue manager name on an MQCONN or MQCONNX call with an asterisk.

You might choose to define connections to more than one server machine because:

- You want to connect a client to any one of a set of queue managers that is running, to improve availability.
- You want to reconnect a client to the same queue manager it connected to successfully last time, but connect to a different queue manager if the connection fails.
- You want to be able to retry a client connection to a different queue manager if the connection fails, by issuing the MQCONN in the client program again.
- You want to automatically reconnect a client connection to another queue manager if the connection fails, without writing any client code.
- You want to automatically reconnect a client connection to a different instance of a multi-instance queue manager if a standby instance takes over, without writing any client code.
- You want to balance your client connections across a number of queue managers, with more clients connecting to some queue managers than others.
- You want to spread the reconnection of many client connections over multiple queue managers and over time, in case the high volume of connections causes a failure.
- You want to be able to move your queue managers without changing any client application code.
- You want to write client application programs that do not need to know queue manager names.

It is not always appropriate to connect to different queue managers. An extended transactional client or a Java client in WebSphere Application Server, for example, might need to connect to a predictable queue manager instance. Automatic client reconnect is not supported by IBM MQ classes for Java.

A queue manager group is a set of connections defined in the client channel definition table (CCDT). The set is defined by its members having the same value of the **QMNAME** attribute in their channel definitions.

Figure 97 on page 891 is a graphical representation of a client connection table, showing three queue manager groups, two named queue manager groups written in the CCDT as **QMNAME** (QM1) and **QMNAME** (QMGrp1), and one blank or default group written as **QMNAME** (' ').

1. Queue manager group QM1 has three client connection channels, connecting it to queue managers QM1 and QM2. QM1 might be a multi-instance queue manager located on two different servers.
2. The default queue manager group has six client connection channels connecting it to all the queue managers.
3. QMGrp1 has client connection channels to two queue managers, QM4 and QM5.

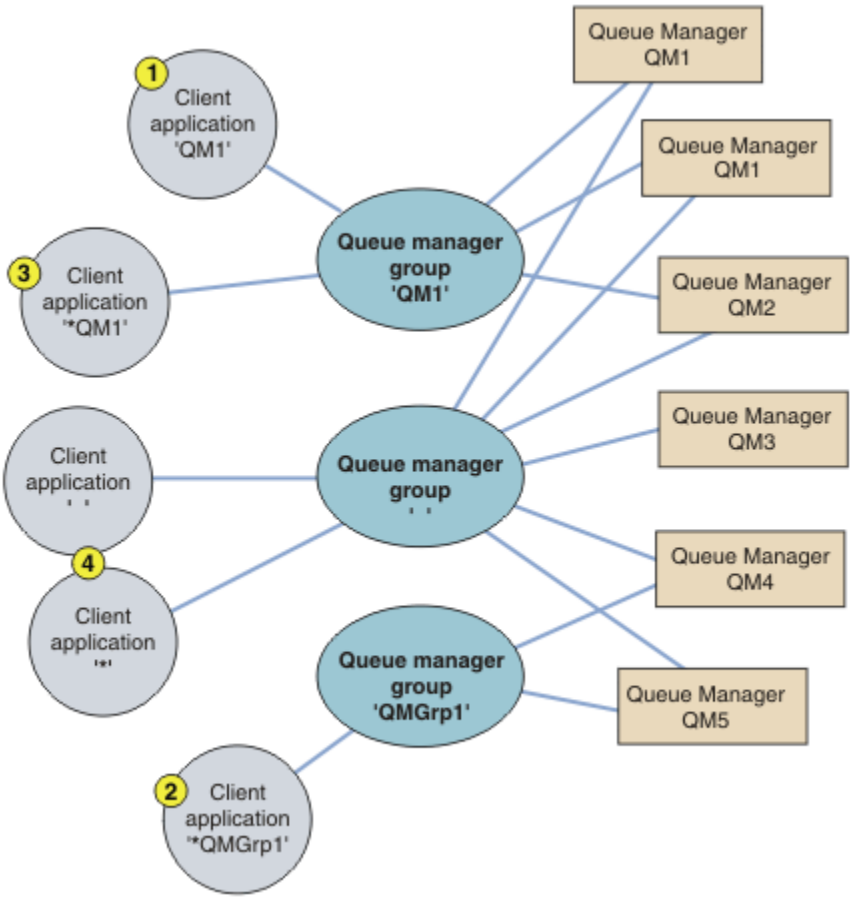


Figure 97. Queue manager groups

Four examples of using this client connection table are described with the help of the numbered client applications in Figure 97 on page 891.

1. In the first example, the client application passes a queue manager name, QM1, as the **QmgrName** parameter to its MQCONN or MQCONNX MQI call. The IBM MQ client code selects the matching queue manager group, QM1. The group contains three connection channels, and the IBM MQ MQI client tries to connect to QM1 using each of these channels in turn until it finds an IBM MQ listener for the connection attached to a running queue manager called QM1.

   The order of connection attempts depends on the value of the client connection AFFINITY attribute and the client channel weightings. Within these constraints, the order of connection attempts is randomized, both over the three possible connections, and over time, in order to spread out the load of making connections.

   The MQCONN or MQCONNX call issued by the client application succeeds when a connection is established to a running instance of QM1.

2. In the second example, the client application passes a queue manager name prefixed with an asterisk, *QMGrp1 as the **QmgrName** parameter to its MQCONN or MQCONNX MQI call. The IBM MQ client selects the matching queue manager group, QMGrp1. This group contains two client connection channels, and the IBM MQ MQI client tries to connect to *any* queue manager using each channel in turn. In this example, the IBM MQ MQI client needs to make a successful connection; the name of the queue manager that it connects to does not matter.

   The rule for the order of making connection attempts is the same as before. The only difference is that by prefixing the queue manager name with an asterisk, the client indicates that the name of the queue manager is not relevant.

The MQCONN or MQCONNX call issued by the client application succeeds when a connection is established to a running instance of any queue manager connected to by the channels in the QMGrp1 queue manager group.

3. The third example is essentially the same as the second because the **QmgrName** parameter is prefixed by an asterisk, *QM1. The example illustrates that you cannot determine which queue manager a client channel connection is going to connect to by inspecting the QMNAME attribute in one channel definition by itself. The fact that the **QMNAME** attribute of the channel definition is QM1, is not sufficient to require a connection is made to a queue manager called QM1. If your client application prefixes its **QmgrName** parameter with an asterisk then any queue manager is a possible connection target.

   In this case the MQCONN or MQCONNX calls issued by the client application succeed when a connection is established to a running instance of either QM1 or QM2.

4. The fourth example illustrates use of the default group. In this case the client application passes an asterisk, '*', or blank ' ', as the **QmgrName** parameter to its MQCONN or MQCONNX MQI call. By convention in the client channel definition, a blank **QMNAME** attribute signifies the default queue manager group and either a blank or asterisk **QmgrName** parameter matches a blank **QMNAME** attribute.

   In this example the default queue manager group has client channel connections to all the queue managers. By selecting the default queue manager group the application might be connected to any queue manager in the group.

   The MQCONN or MQCONNX call issued by the client application succeeds when a connection is established to a running instance of any queue manager.

   Note: The default group is different from a default queue manager, although an application uses a blank **QmgrName** parameter to connect to either the default queue manager group or to the default queue manager. The concept of a default queue manager group is only relevant to a client application, and a default queue manager to a server application.

Define your client connection channels on one queue manager only, including those channels that connect to a second or third queue manager. Do not define them on two queue managers and then try to merge the two client channel definition tables. Only one client channel definition table can be accessed by the client.

Examples

Look again at the list of reasons for using queue manager groups at the beginning of the topic. How does using a queue manager group provide those capabilities?

Connect to any one of a set of queue managers.
Define a queue manager group with connections to all the queue managers in the set, and connect to the group using the **QmgrName** parameter prefixed by an asterisk.

Reconnect to the same queue manager, but connect to a different one, if the queue manager connected to last time is unavailable.
Define a queue manager group as before but set the attribute, **AFFINITY** (PREFERRED) on each client channel definition.

Retry a connection to another queue manager if a connection fails.
Connect to a queue manager group, and reissue the MQCONN or MQCONNX MQI call if the connection is broken or the queue manager fails.

Automatically reconnect to another queue manager if a connection fails.
Connect to a queue manager group using the MQCONNX **MQCNO** option MQCNO_RECONNECT.

Automatically reconnect to a different instance of a multi-instance queue manager.
Do the same as the preceding example. In this case, if you want to restrict the queue manager group to connect to the instances of a particular multi-instance queue manager, define the group with connections to only the multi-instance queue manager instances.

You can also ask the client application to issue its MQCONN or MQCONNX MQI call with no asterisk prefixed to the **QmgrName** parameter. That way the client application can only connect to the named queue manager. Finally, you can set the **MQCNO** option to MQCNO_RECONNECT_Q_MGR. This option

accepts reconnections to the same queue manager that was previously connected. You can also use this value to restrict reconnections to the same instance of a normal queue manager.

Balance client connections across queue managers, with more clients connected to some queue managers than others.
Define a queue manager group, and set the **CLNTWGHT** attribute on each client channel definition to distribute the connections unevenly.

Spread the client reconnection load unevenly, and spread it over time, after a connection or queue manager failure.
Do the same as the preceding example. The IBM MQ MQI client randomizes reconnections across queue managers and spreads the reconnections over time.

Move your queue managers without changing any client code.
The CCDT isolates your client application from the location of the queue manager. The CCDT is a data file which can be defined at the client, read from a shared location or fetched from a web-server. For more information, see Client channel definition table.

Write a client application that does not know queue manager names.
Use queue manager group names and establish a naming convention for queue manager group names that is relevant to your client applications in your organization, and reflects the architecture of your solutions rather than the naming of queue managers.

z/OS *Connecting to queue sharing groups*

You can connect your application to a queue manager that is part of a queue sharing group. This can be done by using the queue sharing group name instead of the queue manager name on the MQCONN or MQCONNX call.

Queue sharing groups have a name of up to four characters. The name must be unique in your network, and must be different from any queue manager names.

The client channel definition should use the queue sharing group generic interface to connect to an available queue manager in the group. For more information, see Connecting a client to a queue sharing group. A check is made to ensure that the queue manager the listener connects to is a member of the queue sharing group.

For more information on shared queues, see Shared queues and queue sharing groups.

Examples of channel weighting and affinity

These examples illustrate how client-connection channels are selected when non-zero `ClientChannelWeights` are used.

The `ClientChannelWeight` and `ConnectionAffinity` channel attributes control how client-connection channels are selected when more than one suitable channel is available for a connection. These channels are configured to connect to different queue managers in order to provide higher availability, workload balancing, or both. MQCONN calls that could result in a connection to one of several queue managers must prefix the queue manager name with an asterisk as described in: Examples of MQCONN calls: Example 1. Queue manager name includes an asterisk (*).

Applicable candidate channels for a connection are those where the QMNAME attribute matches the queue manager name specified in the MQCONN call. If all applicable channels for a connection have a `ClientChannelWeight` of zero (the default) then they are selected in alphabetical order as in the example: Examples of MQCONN calls: Example 1. Queue manager name includes an asterisk (*).

The following examples illustrate what happens when non-zero `ClientChannelWeights` are used. Note that, since this feature involves pseudo-random channel selection, the examples show a sequence of actions that might happen rather than what definitely will.

Example 1. Selecting channels when ConnectionAffinity is set to PREFERRED
This example illustrates how an IBM MQ MQI client selects a channel from a CCDT, where the ConnectionAffinity is set to PREFERRED.

In this example, a number of client machines use a Client Channel Definition Table (CCDT) provided by a queue manager. The CCDT includes client connection channels with the following attributes (shown using the syntax of the DEFINE CHANNEL command):

```
CHANNEL(A) QMNAME(DEV) CONNAME(devqm.it.company.example)
CHANNEL(B) QMNAME(CORE) CONNAME(core1.ops.company.example) CLNTWGHT(5) +
AFFINITY(PREFERRED)
CHANNEL(C) QMNAME(CORE) CONNAME(core2.ops.company.example) CLNTWGHT(3) +
AFFINITY(PREFERRED)
CHANNEL(D) QMNAME(CORE) CONNAME(core3.ops.company.example) CLNTWGHT(2) +
AFFINITY(PREFERRED)
```

The application issues MQCONN(*CORE)

Channel A is not a candidate for this connection, because the QMNAME attribute does not match. Channels B, C and D are identified as candidates, and are placed in an order of preference based on their weighting. In this example the order might be C, B, D. The client attempts to connect to the queue manager at core2.ops.company.example. The name of the queue manager at that address is not checked, because the MQCONN call included an asterisk in the queue manager name.

It is important to note that, with `AFFINITY(PREFERRED)`, each time this particular client machine connects it will place the channels in the same initial order of preference. This applies even when the connections are from different processes or at different times.

In this example, the queue manager at core.2.ops.company.example cannot be reached. The client attempts to connect to core1.ops.company.example because channel B is next in the order of preference. In addition, channel C is demoted to become the least preferred.

A second MQCONN(*CORE) call is issued by the same application. Channel C was demoted by the previous connection, so the most preferred channel is now B. This connection is made to core1.ops.company.example.

A second machine sharing the same Client Channel Definition Table migh place the channels in a different initial order of preference. For example, D, B, C. Under normal circumstances, with all channels working, applications on this machine are connected to core3.ops.company.example while those on the first machine are connected to core2.ops.company.example. This allows workload balancing of large numbers of clients across multiple queue managers while allowing each individual client to connect to the same queue manager if it is available.

Example 2. Selecting channels when ConnectionAffinity is set to NONE
This example illustrates how an IBM MQ MQI client selects a channel from a CCDT, where the ConnectionAffinity is set to NONE.

In this example, a number of clients use a Client Channel Definition Table (CCDT) provided by a queue manager. The CCDT includes client connection channels with the following attributes (shown using the syntax of the DEFINE CHANNEL command):

```
CHANNEL(A) QMNAME(DEV) CONNAME(devqm.it.company.example)
CHANNEL(B) QMNAME(CORE) CONNAME(core1.ops.company.example) CLNTWGHT(5) +
AFFINITY(NONE)
CHANNEL(C) QMNAME(CORE) CONNAME(core2.ops.company.example) CLNTWGHT(3) +
AFFINITY(NONE)
CHANNEL(D) QMNAME(CORE) CONNAME(core3.ops.company.example) CLNTWGHT(2) +
AFFINITY(NONE)
```

The application issues MQCONN(*CORE). As in the previous example, channel A is not considered because the QMNAME does not match. Channel B, C, or D are selected based on their weighting, with probabilities of 50%, 30%, or 20%. In this example, channel B might be selected. There is no persistent order of preference created.

A second MQCONN(*CORE) call is made. Again, one of the three applicable channels is selected, with the same probabilities. In this example, channel C is chosen. However, core2.ops.company.example does not respond, so another choice is made between the remaining candidate channels. Channel B is selected and the application is connected to core1.ops.company.example.

With AFFINITY(NONE), each MQCONN call is independent of any other. Therefore when this example application makes a third MQCONN(*CORE), it might once more attempt to connect through the broken channel C, before choosing one of B or D.

Examples of MQCONN calls
Examples of using MQCONN to connect to a specific queue manager, or to one of a group of queue managers.

In each of the following examples, the network is the same; there is a connection defined to two servers from the same IBM MQ MQI client. (In these examples, the MQCONNX call could be used instead of the MQCONN call.)

There are two queue managers running on the server machines, one named SALE and the other named SALE_BACKUP.

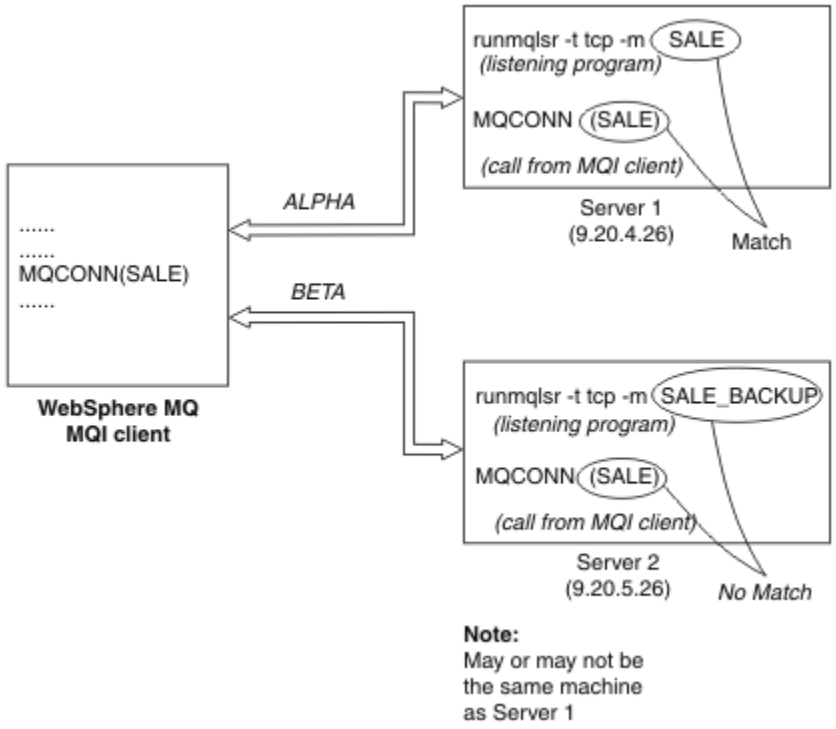


Figure 98. MQCONN example

The definitions for the channels in these examples are:

SALE definitions:

```
DEFINE CHANNEL(ALPHA) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
DESCR('Server connection to IBM MQ MQI client')

DEFINE CHANNEL(ALPHA) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
CONNAME(9.20.4.26) DESCR('IBM MQ MQI client connection to server 1') +
QMNAME(SALE)

DEFINE CHANNEL(BETA) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
CONNAME(9.20.5.26) DESCR('IBM MQ MQI client connection to server 2') +
QMNAME(SALE)
```

SALE_BACKUP definition:

```
DEFINE CHANNEL(BETA) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
DESCR('Server connection to IBM MQ MQI client')
```

The client channel definitions can be summarized as follows:

Name	CHLTYPE	TRPTYPE	CONNAME	QMNAME
ALPHA	CLNTCONN	TCP	9.20.4.26	SALE
BETA	CLNTCONN	TCP	9.20.5.26	SALE

What the MQCONN examples demonstrate
The examples demonstrate the use of multiple queue managers as a backup system.

Suppose the communication link to Server 1 is temporarily broken. The use of multiple queue managers as a backup system is demonstrated.

Each example covers a different MQCONN call and gives an explanation of what happens in the specific example presented, by applying the following rules:

1. The client channel definition table (CCDT) is scanned in alphabetical channel name order for a queue manager name (QMNAME field) corresponding to the one given in the MQCONN call.
2. If a match is found, the channel definition is used.
3. An attempt is made to start the channel to the machine identified by the connection name (CONNAME). If this is successful, the application continues. It requires:
   - A listener to be running on the server.
   - The listener to be connected to the same queue manager as the one the client wants to connect to (if specified).
4. If the attempt to start the channel fails and there is more than one entry in the client channel definition table (in this example there are two entries), the file is searched for a further match. If a match is found, processing continues at step 1.
5. If no match is found, or there are no more entries in the client channel definition table and the channel has failed to start, the application is unable to connect. An appropriate reason code and completion code are returned in the MQCONN call. The application can take action based on the reason and completion codes returned.

Example 1. Queue manager name includes an asterisk (*)
In this example the application is not concerned about which queue manager it connects to. The application issues an MQCONN call for a queue manager name including an asterisk. A suitable channel is chosen.

The application issues:

```
MQCONN (*SALE)
```

Following the rules, this is what happens in this instance:

1. The client channel definition table (CCDT) is scanned for the queue manager name SALE, matching with the application MQCONN call.
2. Channel definitions for ALPHA and BETA are found.
3. If one channel has a CLNTWGHT value of 0, this channel is selected. If both have a CLNTWGHT value of 0, channel ALPHA is selected because it is first in alphabetical sequence. If both channels have a non-zero CLNTWGHT value, one channel is randomly selected, based on its weighting.
4. An attempt to start the channel is made.
5. If channel BETA was selected, the attempt to start it is successful.

6. If channel ALPHA was selected, the attempt to start it is NOT successful because the communication link is broken. The following steps then apply:
   a. The only other channel for the queue manager name SALE is BETA.
   b. An attempt to start this channel is made - this is successful.
7. A check to see that a listener is running shows that there is one running. It is not connected to the SALE queue manager, but because the MQI call parameter has an asterisk (*) included in it, no check is made. The application is connected to the SALE_BACKUP queue manager and continues processing.

Example 2. Queue manager name specified
In this example the application must connect to a particular queue manager. The application issues an MQCONN call for that queue manager name. A suitable channel is chosen.

The application requires a connection to a specific queue manager, named SALE, as seen in the MQI call:

```
MQCONN (SALE)
```

Following the rules, this is what happens in this instance:

1. The client channel definition table (CCDT) is scanned in alphabetical channel name sequence, for the queue manager name SALE, matching with the application MQCONN call.
2. The first channel definition found to match is ALPHA.
3. An attempt to start the channel is made - this is not successful because the communication link is broken.
4. The client channel definition table is again scanned for the queue manager name SALE and the channel name BETA is found.
5. An attempt to start the channel is made - this is successful.
6. A check to see that a listener is running shows that there is one running, but it is not connected to the SALE queue manager.
7. There are no further entries in the client channel definition table. The application cannot continue and receives return code MQRC_Q_MGR_NOT_AVAILABLE.

Example 3. Queue manager name is blank or an asterisk (*)
In this example the application is not concerned about which queue manager it connects to. The application issues an MQCONN specifying a blank queue manager name or an asterisk. A suitable channel is chosen.

This is treated in the same way as “Example 1. Queue manager name includes an asterisk (*)” on page 896.

Note: If this application were running in an environment other than an IBM MQ MQI client, and the name was blank, it would be attempting to connect to the default queue manager. This is not the case when it is run from a client environment; the queue manager accessed is the one associated with the listener to which the channel connects.

The application issues:

```
MQCONN ("")
```

or

```
MQCONN (*)
```

Following the rules, this is what happens in this instance:

1. The client channel definition table (CCDT) is scanned in alphabetical channel name sequence, for a queue manager name that is blank, matching with the application MQCONN call.

2. The entry for the channel name ALPHA has a queue manager name in the definition of SALE. This does not match the MQCONN call parameter, which requires the queue manager name to be blank.
3. The next entry is for the channel name BETA.
4. The `queue manager name` in the definition is SALE. Once again, this does not match the MQCONN call parameter, which requires the queue manager name to be blank.
5. There are no further entries in the client channel definition table. The application cannot continue and receives return code MQRC_Q_MGR_NOT_AVAILABLE.

Triggering in the client environment

Messages sent by IBM MQ applications running on IBM MQ MQI clients contribute to triggering in exactly the same way as any other messages, and they can be used to trigger programs on both the server and the client.

Triggering is explained in detail in the Triggering channels.

The trigger monitor and the application to be started must be on the same system.

The default characteristics of the triggered queue are the same as those in the server environment. In particular, if no MQPMO sync point control options are specified in a client application putting messages to a triggered queue that is local to a z/OS queue manager, the messages are put within a unit of work. If the triggering condition is then met, the trigger message is put on the initiation queue within the same unit of work and cannot be retrieved by the trigger monitor until the unit of work ends. The process that is to be triggered is not started until the unit of work ends.

Process definition

You must define the process definition on the server, because this is associated with the queue that has triggering set on.

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 example, if the trigger monitor is running on an IBM MQ MQI client and wants to send a request to a server on another operating system, MQAT_WINDOWS_NT must be defined otherwise the other operating system uses its default definitions and the process fails.

Multi *Trigger monitor*

The trigger monitor provided by IBM MQ for Multiplatforms runs in the client environments for Multiplatforms systems.

To run the trigger monitor, issue one of these commands:

- IBM i On IBM i:

```
CALL PGM(QMQM/RUNMQTMC) PARM('-m' QmgrName '-q' InitQ)
```

- ALW On AIX, Linux, and Windows platforms:

```
runmqtmc [-m QMgrName] [-q InitQ]
```

The default initiation queue is SYSTEM.DEFAULT.INITIATION.QUEUE on the default queue manager. The initiation queue is where the trigger monitor looks for trigger messages. It then calls programs for the appropriate trigger messages. This trigger monitor supports the default application type and is the same as `runmqtrm` except that it links the client libraries.

The command string, built by the trigger monitor, is as follows:

1. The *ApplicId* from the relevant process definition. *ApplicId* is the name of the program to run, as it would be entered on the command line.

2. The MQTMC2 structure, enclosed in quotation marks, obtained from the initiation queue. A command string is started that has this string, exactly as provided, in quotation marks in order that the system command accepts it as one parameter.
3. The *EnvrData* from the relevant process definition.

The trigger monitor does not look to see if there is another message on the initiation queue until the completion of the application it has started. If the application has much processing to do, the trigger monitor might not keep up with the number of trigger messages arriving. There are two ways to deal with this situation:

1. Have more trigger monitors running

   If you choose to have more trigger monitors running, you can control the maximum number of applications that can run at any one time.
2. Run the started applications in the background

   If you choose to run applications in the background, IBM MQ imposes no restriction on the number of applications that can run.

To run the started application in the background on AIX and Linux systems, you must put an & (ampersand) at the end of the *EnvrData* of the process definition.

CICS applications (non-z/OS)
A non-z/OS CICS application program that issues an MQCONN or MQCONNX call must be defined to CEDA as RESIDENT. If you relink a CICS server application as a client, you risk losing sync point support.

A non-z/OS CICS application program that issues an MQCONN or MQCONNX call must be defined to CEDA as RESIDENT. To make the resident code as small as possible, you can link to a separate program to issue the MQCONN or MQCONNX call.

If the MQSERVER environment variable is used to define the client connection, it must be specified in the CICSENV.CMD file.

IBM MQ applications can be run in an IBM MQ server environment or on an IBM MQ client without changing code. However, in an IBM MQ server environment, CICS can act as sync point coordinator, and you use EXEC CICS SYNCPOINT and EXEC CICS SYNCPOINT ROLLBACK rather than **MQCMIT** and **MQBACK**. If a CICS application is simply relinked as a client, sync point support is lost. **MQCMIT** and **MQBACK** must be used for the application running on an IBM MQ MQI client.

ALW Preparing and running CICS and Tuxedo applications

To run CICS and Tuxedo applications as client applications, you use different libraries from those you use with server applications. The user ID under which the application runs is also different.

To prepare CICS and Tuxedo applications to run as IBM MQ MQI client applications, follow the instructions in the Configuring an extended transactional client.

Note, however, that the information that deals specifically with preparing CICS and Tuxedo applications, including the sample programs supplied with IBM MQ, assumes that you are preparing applications to run on an IBM MQ server system. As a result, the information refers only to IBM MQ libraries that are intended for use on a server system. When you are preparing your client applications, you must do the following things:

- Use the appropriate client system library for the language bindings that your application uses. For example:
  - Linux AIX For applications written in C on AIX and Linux, use the library libmqic instead of libmqm.
  - Windows On Windows systems, use the library mqic.lib instead of mqm.lib.
- Instead of the server system libraries shown in Table 134 on page 900 and Table 135 on page 900, use the equivalent client system libraries. If a server system library is not listed in these tables, use the same library on a client system.

Table 134. Client system libraries on AIX and Linux	
Library for an IBM MQ server system	**Equivalent library to use on an IBM MQ client system**
libmqmxa	libmqcxa
V 9.4.0 V 9.4.0 libmqmxa64	V 9.4.0 V 9.4.0 libmqcxa64

Table 135. Client system libraries on Windows systems	
Library for an IBM MQ server system	**Equivalent library to use on an IBM MQ client system**
mqmxa.lib	mqcxa.lib
mqmtux.lib	mqcxa.lib
mqmenc.lib	mqcxa.lib
mqmcics4.lib	mqccics4.lib

The user ID used by a client application

When you run an IBM MQ server application under CICS, it normally switches from the CICS user to the user ID of the transaction. However, when you run an IBM MQ MQI client application under CICS, it retains the CICS privileged authority.

ALW *CICS and Tuxedo sample programs*

CICS and Tuxedo sample programs for use on AIX, Linux, and Windows systems.

Table 136 on page 900 lists the CICS and Tuxedo sample programs that are supplied for use on AIX and Linux client systems. Table 137 on page 900 lists the equivalent information for Windows client systems. The tables also list the files that are used for preparing and running the programs. For a description of the sample programs, see "The CICS transaction sample" on page 1040 and "Using the TUXEDO samples on AIX, Linux, and Windows" on page 1083.

Table 136. Sample programs for AIX and Linux client systems		
Description	**Source**	**Executable module**
CICS program	amqscic0.ccs	amqscicc
Header file for the CICS program	amqscih0.h	-
Tuxedo client program to put messages	amqstxpx.c	-
Tuxedo client program to get messages	amqstxgx.c	-
Tuxedo server program for the two client programs	amqstxsx.c	-
UBBCONFIG file for the Tuxedo programs	ubbstxcx.cfg	-
Field table file for the Tuxedo programs	amqstxvx.flds	-
View description file for the Tuxedo programs	amqstxvx.v	-

Table 137. Sample programs for Windows client systems		
Description	**Source**	**Executable module**
CICS transaction	amqscic0.ccs	amqscicc

Table 137. Sample programs for Windows client systems (continued)		
Description	**Source**	**Executable module**
Header file for the CICS transaction	amqscih0.h	-
Tuxedo client program to put messages	amqstxpx.c	-
Tuxedo client program to get messages	amqstxgx.c	-
Tuxedo server program for the two client programs	amqstxsx.c	-
UBBCONFIG file for the Tuxedo programs	ubbstxcx.cfg	-
Field table file for the Tuxedo programs	amqstxvx.fld	-
View description file for the Tuxedo programs	amqstxvx.v	-
Makefile for the Tuxedo programs	amqstxmc.mak	-
ENVFILE file for the Tuxedo programs	amqstxen.env	-

ALW *Error message AMQ5203, as modified for CICS and Tuxedo applications*

When you run CICS or Tuxedo applications that use an extended transactional client, you might see standard diagnostic messages. One of these has been modified for use with an extended transactional client

The messages that you might see in the IBM MQ error log files are documented in Diagnostic messages: AMQ4000-9999. Message AMQ5203 has been modified for use with an extended transactional client. Here is the text of the modified message:

AMQ5203: An error occurred calling the XA interface.

Explanation

The error number is &2 where a value of 1 indicates the supplied flags value of &1 was invalid, 2 indicates that there was an attempt to use threaded and non-threaded libraries in the same process, 3 indicates that there was an error with the supplied queue manager name '&3', 4 indicates that the resource manager ID of &1 was invalid, 5 indicates that an attempt was made to use a second queue manager called '&3' when another queue manager was already connected, 6 indicates that the Transaction Manager has been called when the application isn't connected to a queue manager, 7 indicates that the XA call was made while another call was in progress, 8 indicates that the xa_info string '&4' in the xa_open call contained an invalid parameter value for parameter name '&5', and 9 indicates that the xa_info string '&4' in the xa_open call is missing a required parameter, parameter name '&5'.

User response

Correct the error and try the operation again.

Windows Preparing and running Microsoft Transaction Server applications

To prepare an MTS application to run as an IBM MQ MQI client application, follow these instructions as appropriate for your environment.

For general information about how to develop Microsoft Transaction Server (MTS) applications that access IBM MQ resources, see the section on MTS in the IBM MQ Help Center.

To prepare an MTS application to run as an IBM MQ MQI client application, do one of the following for each component of the application:

- If the component uses the C language bindings for the MQI, follow the instructions in “Preparing C programs in Windows” on page 978 but link the component with the library mqicxa.lib instead of mqic.lib.

- If the component uses the IBM MQ C++ classes, follow the instructions in "Building C++ programs on Windows" on page 531 but link the component with the library imqx23vn.lib instead of imqc23vn.lib.
- If the component uses the Visual Basic language bindings for the MQI, follow the instructions in the "Preparing Visual Basic programs in Windows" on page 981 but when you define the Visual Basic project, type `MqType=3` in the **Conditional Compilation Arguments** field.

Preparing and running IBM MQ JMS applications

You can run IBM MQ JMS applications in client mode, with WebSphere Application Server as your transaction manager. You might see certain warning messages.

To prepare and run IBM MQ JMS applications in client mode, with WebSphere Application Server as your transaction manager, follow the instructions in "Using IBM MQ classes for JMS/Jakarta Messaging" on page 79.

When you run an IBM MQ JMS client application, you might see the following warning messages:

MQJE080
Insufficient license units - run setmqcap

MQJE081
File containing the license unit information is in the wrong format - run setmqcap

MQJE082
File containing the license unit information could not be found - run setmqcap

User exits, API exits, and IBM MQ installable services

This topic contains links to information about using and developing these programs.

For an introduction to how you can use user exits, API exits, and installable services to extend queue manager facilities, see Extending queue manager facilities.

For information on writing and compiling exits and installable services, see the subtopics.

Related concepts
Channel-exit programs for MQI channels
Related reference
API exit reference
Installable services interface reference information
IBM i Installable services interface reference information on IBM i

ALW Writing exits and installable services on AIX, Linux, and Windows

You can write and compile exits without linking to any IBM MQ libraries on AIX, Linux, and Windows.

About this task

This topic applies to AIX, Linux, and Windows systems only. For details on writing exits and installable services for other platforms, see the relevant platform specific topics.

If IBM MQ is installed in a non-default location you must write and compile your exits without linking to any IBM MQ libraries.

You can write and compile exits on AIX, Linux, and Windows systems without linking any of these IBM MQ libraries:

- mqmzf
- mqm
- mqmvx
- mqmvxd

- mqic
- mqutl

Existing exits that are linked to these libraries continue to work, providing that on AIX and Linux systems IBM MQ is installed in the default location.

Procedure

1. Include the cmqec.h header file.

   Including this header file automatically includes the cmqc.h, cmqxc.h and cmqzc.h header files.
2. Write the exit so that MQI and DCI calls are made through the MQIEP structure. For more information about the MQIEP structure, see MQIEP structure.
   - Installable services
     - Use the **Hconfig** parameter to point to the MQZEP call.
     - You must check that the first 4 bytes of **Hconfig** match the **StrucId** of the MQIEP structure before using the **Hconfig** parameter.
     - For more information about writing installable service components, see MQIEP.
   - API exits
     - Use the **Hconfig** parameter to point to the MQXEP call.
     - You must check that the first 4 bytes of **Hconfig** match the **StrucId** of the MQIEP structure before using the **Hconfig** parameter.
     - For more information about writing API exits, see "Writing API exits" on page 919.
   - Channel exits
     - Use the **pEntryPoints** parameter of the MQCXP structure to point to MQI and DCI calls.
     - You must check that the MQCXP version number is at version 8 or higher before using **pEntryPoints**.
     - For more information about writing channel exits, see "Writing channel-exit programs" on page 929.
   - Data conversion exits
     - Use the **pEntryPoints** parameter of the MQDXP structure to point to MQI and DCI calls.
     - You must check that the MQDXP version number is at version 2 or higher before using **pEntryPoints**.
     - You can use the **crtmqcvx** command and the amqsvfc0.c source file to create data conversion code that uses the **pEntryPoints** parameter. See "Writing a data-conversion exit for IBM MQ for Windows" on page 954 and "Writing a data-conversion exit for IBM MQ for AIX or Linux systems" on page 951.
     - If you have existing data conversion exits that were generated using the **crtmqcvx** command, you must regenerate the exit using the updated command.
     - For more information about writing data conversion exits, see "Writing data-conversion exits" on page 947.
   - Pre-connect exits
     - Use the **pEntryPoints** parameter of the MQNXP structure to point to MQI and DCI calls.
     - You must check that the MQNXP version number is at version 2 or higher before using **pEntryPoints**.
     - For more information about writing pre-connect exits, see "Referencing connection definitions using a pre-connect exit from a repository" on page 956.
   - Publish exits
     - Use the **pEntryPoints** parameter of the MQPSXP structure to point to MQI and DCI calls.

- You must check that the MQPSXP version number is at version 2 or higher before using **pEntryPoints**.
- For more information about writing publish exits, see "Writing and compiling publish exits" on page 957.

- Cluster workload exits
  - Use the **pEntryPoints** parameter of the MQWXP structure to point to MQXCLWLN calls.
  - You must check that the MQWXP version number is at version 4 or higher before using **pEntryPoints**.
  - For more information about writing cluster workload exits, see "Writing and compiling cluster workload exits" on page 959.

For example, in a channel exit calling MQPUT:

```
pChannelExitParms -> pEntryPoints -> MQPUT_Call(pChannelExitParms -> Hconn,
                                                Hobj,
                                                &md,
                                                &pmo,
                                                messlen,
                                                buffer,
                                                &CompCode,
                                                &Reason);
```

Further examples can be seen in the "Using the IBM MQ sample procedural programs" on page 1018.

3. Compile the exit:
   - Do not link to the IBM MQ libraries.
   - Do not include an embedded RPath to any IBM MQ libraries in your exit.
   - For more information about compiling your exit, see one of the following topics:
     - API exits: "Compiling API exits" on page 921.
     - Channel exits, publish exits, Cluster workload exits: "Compiling channel exit programs on AIX, Linux, and Windows systems" on page 946.
     - Data conversion exits: "Writing data-conversion exits" on page 947.
4. Put the exit in one of the following places:
   - A path of your choosing that you fully qualify when configuring the exit
   - The default exit path, in a specific installation directory. For example, *MQ_DATA_PATH*`/exits/installation2`.
   - The default exit path

     The default exit path is *MQ_DATA_PATH*`/exits` for 32 bit exits, and *MQ_DATA_PATH*`/exits64` for 64 bit exits. You can change these paths in the qm.ini or mqclient.ini file. For more information, see Exit path. On Windows and Linux, you can use the IBM MQ Explorer to change the path:

     a. Right click the queue manager name

     b. Click **Properties...**

     c. Click **Exits**

     d. In the exits default path field, specify the path name of the directory that holds the exit program.

   If an exit is placed in both a specific installation directory and in the default path directory, the specific installation directory exit is used by the installation of IBM MQ named in the path. For example, the exit is placed in `/exits/installation2` and in `/exits`, but not in `/exits/installation1`. The IBM MQ installation `installation2` uses the exit from `/exits/installation2`. The IBM MQ installation `installation1` uses the exit from the `/exits` directory.
5. If necessary, configure the exit:
   - Installable services: "Configuring services and components" on page 912.

- API exits: “Configuring API exits” on page 923.
- Channel exits: “Configuring channel exits” on page 947.
- Publish exits: “Configuring publish exits” on page 959.
- Pre-connect exits: PreConnect stanza of the client configuration file.

ALW *API exits not linked with an MQI library*

Under certain circumstances, you should link your existing API exit, that cannot be re-coded to use the MQIEP function pointers, with an IBM MQ API library.

This is necessary, so that your existing API exit can be successfully loaded, by the runtime linker of your system, into programs that do not already have the function pointers loaded.

Note: This information is limited to those existing API exits that make MQI calls directly. That is, those exits that do not use , MQIEP. Where possible, you should plan to re-code the exit to use the MQIEP entry points instead.

runmqsc is an example of a program that does not link directly with an MQI library.

Therefore, an API exit that has not been linked with its required IBM MQ API library, or re-coded to use the MQIEP, fails to load into **runmqsc**.

You see errors in the queue manager error log, for example, AMQ6175: `The system could not dynamically load the shared library`, together with qualifying text such as `undefined symbol: MQCONN`.

and AMQ7214: `The module for API Exit 'myexitname' could not be loaded.`

Related tasks

“Writing exits and installable services on AIX, Linux, and Windows” on page 902
You can write and compile exits without linking to any IBM MQ libraries on AIX, Linux, and Windows.

ALW Installable services and components for AIX, Linux, and Windows

This section introduces the installable services and the functions and components associated with them. The interface to these functions is documented so that you, or software vendors, can supply components.

The main reasons for providing IBM MQ installable services are:

- To provide you with the flexibility of choosing whether to use components provided by IBM MQ products, or replace or augment them with others.
- To allow vendors to participate, by providing components that might use new technologies, without making internal changes to IBM MQ products.
- To allow IBM MQ to exploit new technologies faster and cheaper, and so provide products earlier and at lower prices.

Installable services and *service components* are part of the IBM MQ product structure. At the center of this structure is the part of the queue manager that implements the function and rules associated with the Message Queue Interface (MQI). This central part requires a number of service functions, called *installable services*, in order to perform its work. The installable services are:

- Authorization service
- Name service

Each installable service is a related set of functions implemented using one or more *service components*. Each component is invoked using a properly-architected, publicly-available interface. This enables independent software vendors and other third parties to provide installable components to augment or replace those provided by the IBM MQ products. Table 138 on page 906 summarizes the services and components that can be used.

Table 138. Installable service components summary			
Installable service	**Supplied component**	**Function**	**Requirements**
Authorization service	object authority manager (OAM)	Provides authorization checking on commands and MQI calls. Users can write their own component to augment or replace the OAM. For example, to check that a user ID has authority to open a queue.	(Appropriate platform authorization facilities are assumed)
Name service	None	Provides support to the queue manager for looking up the name of the queue manager that owns a specified queue. • User defined	• A third-party or user-written name manager

The installable services interface is described in Installable services interface reference information.

Related tasks
Configuring installable services

Writing a service component

This section describes the relationship between services, components, entry points, and return codes.

Functions and components

Each service consists of a set of related functions. For example, the name service contains function for:

- Looking up a queue name and returning the name of the queue manager where the queue is defined
- Inserting a queue name into the service's directory
- Deleting a queue name from the service's directory

It also contains initialization and termination functions.

An installable service is provided by one or more service components. Each component can perform some or all of the functions that are defined for that service. For example, in IBM MQ for AIX, the supplied authorization service component, the OAM, performs all the available functions. See “Authorization service interface” on page 909 for more information. The component is also responsible for managing any underlying resources or software (for example, an LDAP directory) that it needs to implement the service. Configuration files provide a standard way of loading the component and determining the addresses of the functional routines that it provides.

Figure 99 on page 907 shows how services and components are related:

- A service is defined to a queue manager by stanzas in a configuration file.
- Each service is supported by supplied code in the queue manager. Users cannot change this code and therefore cannot create their own services.
- Each service is implemented by one or more components; these can be supplied with the product or user-written. Multiple components for a service can be invoked, each supporting different facilities within the service.
- Entry points connect the service components to the supporting code in the queue manager.

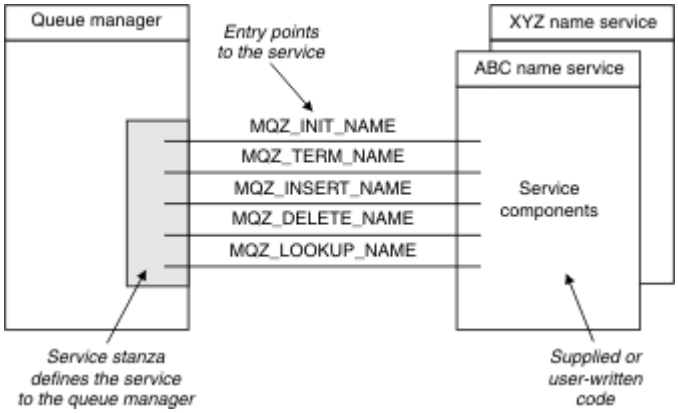


Figure 99. Understanding services, components, and entry points

Entry-points

Each service component is represented by a list of the entry-point addresses of the routines that support a particular installable service. The installable service defines the function to be performed by each routine.

The ordering of the service components when they are configured defines the order in which entry-points are called in an attempt to satisfy a request for the service.

In the supplied header file `cmqzc.h`, the supplied entry points to each service have an MQZID_ prefix.

If the services are present, the services are loaded in a predefined order. The following list shows the services, and the order in which they are initialized.

1. `NameService`
2. `AuthorizationService`
3. `UserIdentifierService`

The `AuthorizationService` is the only service that is configured by default. Configure the `NameService` and `UserIdentifierService` manually if you want to use them.

Services, and service components have a one-to-one or one-to-many mapping. Multiple service components can be defined for each service. On AIX and Linux systems, the `ServiceComponent` stanza's Service value must match the Service stanza's Name value in the qm.ini file. On Windows, the `ServiceComponent` 's Service registry key value must match the Name registry key value, and is defined as: `HKEY_LOCAL_MACHINE\SOFTWARE\IBM\WebSphere MQ\Installation\`*`MQ_INSTALLATION_NAME`*`\Configuration\QueueManager\`*`qmname`*`\` where *qmname* is the name of the queue manager.

For AIX and Linux systems, service components are started in the order they are defined in the qm.ini file. On Windows, because the Windows registry is used, IBM MQ issues a **RegEnumKey** call which returns the values in alphabetic order. Therefore, on Windows the services are called in alphabetic order, as they are defined in the registry.

The ordering of the `ServiceComponent` definitions is significant. This ordering dictates the order in which components are run for a given service. For example, the `AuthorizationService` on Windows is configured with the default OAM component named `MQSeries.WindowsNT.auth.service`. Additional components can be defined for this service in order to override the default OAM. Unless `MQCACF_SERVICE_COMPONENT` is specified, the first component encountered in alphabetic order is used to process the request, and the name for that component is used.

Return codes

Service components provide return codes to the queue manager to report on various conditions. They report the success or failure of the operation, and indicate whether the queue manager is to proceed to the next service component. A separate *Continuation* parameter carries this indication.

Component data

A single service component might require data to be shared between its various functions. Installable services provide an optional data area to be passed on each invocation of a service component. This data area is for the exclusive use of the service component. It is shared by all the invocations of a particular function, even if they are made from different address spaces or processes. It is guaranteed to be addressable from the service component whenever it is called. You must declare the size of this area in the *ServiceComponent* stanza.

Initialization and termination of components
The use of component initialization and termination options.

When the component initialization routine is invoked, it must call the queue manager **MQZEP** function for each entry-point supported by the component. **MQZEP** defines an entry-point to the service. All the undefined exit points are assumed to be NULL.

A component is always invoked once with the primary initialization option, before it is invoked in any other way.

A component can be invoked with the secondary initialization option on certain platforms. For example, it can be invoked once for each operating system process, thread, or task by which the service is accessed.

If secondary initialization is used:

- The component can be invoked more than once for secondary initialization. For each such call, a matching call for secondary termination is issued when the service is no longer needed.

  For naming services this is the MQZ_TERM_NAME call.

  For authorization services this is the MQZ_TERM_AUTHORITY call.
- The entry points must be re-specified (by calling MQZEP) each time the component is called for primary and secondary initialization.
- Only one copy of component data is used for the component; there is not a different copy for each secondary initialization.
- The component is not invoked for any other calls to the service (from the operating system process, thread, or task, as appropriate) before secondary initialization has been carried out.
- The component must set the **Version** parameter to the same value for primary and secondary initialization.

The component is always invoked with the primary termination option once, when it is no longer required. No further calls are made to this component.

The component is invoked with the secondary termination option, if it has been invoked for secondary initialization.

Object authority manager (OAM)
The authorization service component supplied with the IBM MQ products is called the Object Authority Manager (OAM).

By default, the OAM is active and works with the control commands **dspmqaut** (display authority), **dmpmqaut** (dump authority), and **setmqaut** (set or reset authority).

The syntax of these commands and how to use them are described in Administering IBM MQ for Multiplatforms using control commands.

The OAM works with the *entity* of a principal or group:

- Linux AIX On AIX and Linux systems, a principal is a user ID, or an ID associated with an application program running on behalf of a user; a group is a system-defined collection of principals.
- Windows On Windows systems, a principal is a Windows user ID, or an ID associated with an application program running on behalf of a user; a group is a Windows group.

Authorizations can be granted or revoked at the principal or group level.

When an MQI request is made or a command is issued, the OAM checks whether the entity associated with the operation has authorization to perform the requested operation and to access the specified queue manager resources.

The authorization service enables you to augment or replace the authority checking provided for queue managers by writing your own authorization service component.

Name service
The name service is an installable service that provides support to the queue manager for looking up the name of the queue manager that owns a specified queue. No other queue attributes can be retrieved from a name service.

The name service enables an application to open remote queues for output as if they were local queues. A name service is not invoked for objects other than queues.

Note: The remote queues must have their **Scope** attribute set to CELL.

When an application opens a queue, it looks for the name of the queue first in the queue manager's directory. If it does not find it there, it looks in as many name services as have been configured, until it finds one that recognizes the queue name. If none recognizes the name, the open fails.

The name service returns the owning queue manager for that queue. The queue manager then continues with the MQOPEN request as if the command had specified the queue and queue manager name in the original request.

The name service interface (NSI) is part of the IBM MQ framework.

How the name service works

If a queue definition specifies the **Scope** attribute as queue manager, that is, SCOPE(QMGR) in MQSC, the queue definition (along with all the queue attributes) is stored in the queue manager's directory only. This cannot be replaced by an installable service.

If a queue definition specifies the **Scope** attribute as cell, that is, SCOPE(CELL) in MQSC, the queue definition is again stored in the queue manager's directory, along with all the queue attributes. However, the queue and queue manager name are also stored in a name service. If no service is available that can store this information, a queue with the *Scope* cell cannot be defined.

The directory in which the information is stored can be managed by the service, or the service can use an underlying service, for example, an LDAP directory, for this purpose. In either case, definitions stored in the directory must persist, even after the component and queue manager have terminated, until they are explicitly deleted.

Note:

1. To send a message to a remote host's local queue definition (with a scope of CELL) on a different queue manager within a naming directory cell, you need to define a channel.
2. You cannot get messages directly from the remote queue, even when it has a scope of CELL.
3. No remote queue definition is required when sending to a queue with a scope of CELL.
4. The naming service centrally defines the destination queue, although you still need a transmission queue to the destination queue manager and a pair of channel definitions. In addition, the transmission queue on the local system must have the same name as the queue manager owning the target queue, with the scope of cell, on the remote system.

   For example, if the remote queue manager has the name QM01, the transmission queue on the local system must also have the name QM01.

Authorization service interface
The authorization service provides entry points for use by the queue manager.

The entry points are as follows:

MQZ_AUTHENTICATE_USER
Authenticates a user ID and password, and can set identity context fields.

MQZ_CHECK_AUTHORITY
Checks whether an entity has authority to perform one or more operations on a specified object.

MQZ_CHECK_PRIVILEGED
Checks whether a specified user is a privileged user.

MQZ_COPY_ALL_AUTHORITY
Copies all the current authorizations that exist for a referenced object to another object.

MQZ_DELETE_AUTHORITY
Deletes all authorizations associated with a specified object.

MQZ_ENUMERATE_AUTHORITY_DATA
Retrieves all the authority data that matches the selection criteria specified.

MQZ_FREE_USER
Frees associated allocated resources.

MQZ_GET_AUTHORITY
Gets the authority that an entity has to access a specified object.

MQZ_GET_EXPLICIT_AUTHORITY
Gets either the authority that a named group has to access a specified object (but without the additional authority of the **nobody** group) or the authority that the primary group of the named principal has to access a specified object.

MQZ_INIT_AUTHORITY
Initializes authorization service component.

MQZ_INQUIRE
Queries the supported functionality of the authorization service.

MQZ_REFRESH_CACHE
Refresh all authorizations.

MQZ_SET_AUTHORITY
Sets the authority that an entity has to a specified object.

MQZ_TERM_AUTHORITY
Terminates authorization service component.

In addition, on IBM MQ for Windows, the authorization service provides the following entry points for use by the queue manager:

- **MQZ_CHECK_AUTHORITY_2**
- **MQZ_GET_AUTHORITY_2**
- **MQZ_GET_EXPLICIT_AUTHORITY_2**
- **MQZ_SET_AUTHORITY_2**

These entry points support the use of the Windows Security Identifier (NT SID).

These names are defined as **typedef** s, in the header file `cmqzc.h`, which can be used to prototype the component functions.

The initialization function (**MQZ_INIT_AUTHORITY**) must be the main entry point for the component. The other functions are invoked through the entry point address that the initialization function has added into the component entry point vector.

Name service interface
A name service provides entry points for use by the queue manager.

The following entry points are provided:

MQZ_INIT_NAME
Initialize the name service component.

MQZ_TERM_NAME
Terminate the name service component.

MQZ_LOOKUP_NAME
Look up the queue manager name for the specified queue.

MQZ_INSERT_NAME
Insert an entry containing the owning queue manager name for the specified queue into the directory used by the service.

MQZ_DELETE_NAME
Delete the entry for the specified queue from the directory used by the service.

If there is more than one name service configured:

- For lookup, the MQZ_LOOKUP_NAME function is invoked for each service in the list until the queue name is resolved (unless any component indicates that the search should stop).
- For insert, the MQZ_INSERT_NAME function is invoked for the first service in the list that supports this function.
- For delete, the MQZ_DELETE_NAME function is invoked for the first service in the list that supports this function.

Do not have more than one component that supports the insert and delete functions. However, a component that only supports lookup is feasible, and could be used, for example, as the last component in the list to resolve any name that is not known by any other name service component to a queue manager at which the name can be defined.

In the C programming language the names are defined as function data types using the typedef statement. These can be used to prototype the service functions, to ensure that the parameters are correct.

The header file that contains all the material specific to installable services is `cmqzc.h` for the C language.

Apart from the initialization function (MQZ_INIT_NAME), which must be the component's main entry point, functions are invoked by the entry point address that the initialization function has added, using the MQZEP call.

Using multiple service components
You can install more than one component for a service. This allows components to provide only partial implementations of the service, and to rely on other components to provide the remaining functions.

Example of using multiple components

Suppose you create two a name services components called `ABC_name_serv` and `XYZ_name_serv`.

`ABC_name_serv`
This component supports inserting a name in, or deleting a name from, the service directory, but does not support looking up a queue name.

`XYZ_name_serv`
This component supports looking up a queue name, but does not support inserting a name in, or deleting a name from, the service directory.

Component `ABC_name_serv` holds a database of queue names, and uses two simple algorithms to either insert, or delete, a name from the service directory.

Component `XYZ_name_serv` uses a simple algorithm that returns a fixed queue manager name for any queue name with which it is invoked. It does not hold a database of queue names, and therefore does not support the insert and delete functions.

The components are installed on the same queue manager. The *`ServiceComponent`* stanzas are ordered so that component `ABC_name_serv` is invoked first. Any calls to insert or delete a queue in a component directory are handled by component `ABC_name_serv` ; it is the only one that implements these functions. However, a lookup call that component `ABC_name_serv` cannot resolve is passed on to

the lookup-only component, XYZ_name_serv. This component supplies a queue manager name from its simple algorithm.

Omitting entry points when using multiple components

If you decide to use multiple components to provide a service, you can design a service component that does not implement certain functions. The installable services framework places no restrictions on which you can omit. However, for specific installable services, omission of one or more functions might be logically inconsistent with the purpose of the service.

Example of entry points used with multiple components

Table 139 on page 912 shows an example of the installable name service for which the two components have been installed. Each supports a different set of functions associated with this particular installable service. For insert function, the ABC component entry-point is invoked first. Entry points that have not been defined to the service (using **MQZEP**) are assumed to be NULL. An entry-point for initialization is provided in the table, but this is not required because initialization is carried out by the main entry-point of the component.

When the queue manager has to use an installable service, it uses the entry-points defined for that service (the columns in Table 139 on page 912). Taking each component in turn, the queue manager determines the address of the routine that implements the required function. It then calls the routine, if it exists. If the operation is successful, any results and status information are used by the queue manager.

Table 139. Example of entry-points for an installable service

Function number	ABC name service component	XYZ name service component
MQZID_INIT_NAME (Initialize)	ABC_initialize()	XYZ_initialize()
MQZID_TERM_NAME (Terminate)	ABC_terminate()	XYZ_terminate()
MQZID_INSERT_NAME (Insert)	ABC_Insert()	NULL
MQZID_DELETE_NAME (Delete)	ABC_Delete()	NULL
MQZID_LOOKUP_NAME (Lookup)	NULL	XYZ_Lookup()

If the routine does not exist, the queue manager repeats this process for the next component in the list. In addition, if the routine does exist but returns a code indicating that it could not perform the operation, the attempt continues with the next available component. Routines in service components might return a code that indicates that no further attempts to perform the operation should be made.

Configuring services and components

You configure service components using the queue manager configuration files, except on Windows systems, where each queue manager has its own stanza in the Registry.

Procedure

1. Add stanzas to the queue manager configuration file, `qm.ini`, to define the service to the queue manager and specify the location of the module:
   - Each service used must have a `Service` stanza, which defines the service to the queue manager. For more information, see Service stanza of the qm.ini file.
   - For each component within a service, there must be a `ServiceComponent` stanza. This stanza identifies the name and path of the module containing the code for that component. For more information, see ServiceComponent stanza of the qm.ini file.

   The authorization service component, known as the Object Authority Manager (OAM), is supplied with the product. When you create a queue manager, the queue manager configuration file (or the Registry on Windows systems) is automatically updated to include the appropriate stanzas for the authorization

service and for the default component (the OAM). For the other components, you must configure the queue manager configuration file manually.

The code for each service component is loaded into the queue manager when the queue manager is started, using dynamic binding, where this is supported on the platform.

2. Stop and restart the queue manager to activate the component.

Related reference

Service stanza of the qm.ini file

ServiceComponent stanza of the qm.ini file

Refreshing the OAM after changing a user's authorization

In IBM MQ, you can refresh the OAM's authorization group information immediately after changing a user's authorization group membership, reflecting changes made at the operating system level, without needing to stop and restart the queue manager. In order to do this, issue the **REFRESH SECURITY** command.

Note: When you change authorizations with the `setmqaut` command, the OAM implements such changes immediately.

Queue managers store authorization data on a local queue called SYSTEM.AUTH.DATA.QUEUE. This data is managed by **amqzfuma.exe**.

Related reference

REFRESH SECURITY

IBM i Installable services and components on IBM i

Use this information to learn about the installable services and the functions and components associated with them. The interface to these functions is documented so that you, or software vendors, can supply components.

The main reasons for providing IBM MQ installable services are:

- To provide you with the flexibility of choosing whether to use components provided by IBM MQ for IBM i, or replace or augment them with others.
- To allow vendors to participate, by providing components that might use new technologies, without making internal changes to IBM MQ for IBM i.
- To allow IBM MQ to exploit new technologies faster and cheaper, and so provide products earlier and at lower prices.

Installable services and *service components* are part of the IBM MQ product structure. At the center of this structure is the part of the queue manager that implements the function and rules associated with the Message Queue Interface (MQI). This central part requires a number of service functions, called *installable services*, in order to perform its work. The installable service available in IBM MQ for IBM i is the authorization service.

Each installable service is a related set of functions implemented using one or more *service components*. Each component is invoked using a properly-architected, publicly-available interface. This enables independent software vendors and other third parties to provide installable components to augment or replace those provided by IBM MQ for IBM i. Table 140 on page 913 summarizes support for the authorization service.

Table 140. Authorization service components summary

Supplied component	Function	Requirements
Object authority manager (OAM)	Provides authorization checking on commands and MQI calls. Users can write their own component to augment or replace the OAM.	(Appropriate platform authorization facilities are assumed)

Table 140. Authorization service components summary (continued)		
Supplied component	**Function**	**Requirements**
DCE name service component **Note:** DCE is only supported on versions of IBM MQ earlier than V6.0.	• Allows queue managers to share queues, or • User defined **Note:** Shared queues must have their **Scope** attribute set to CELL.	• DCE is required for the supplied component, or • A third-party or user-written name manager

IBM i *Functions and components on IBM i*

Use this information to understand the functions and components, entry-points, return codes, and component data that you can use in IBM MQ for IBM i.

Each service consists of a set of related functions. For example, the name service contains function for:

- Looking up a queue name and returning the name of the queue manager where the queue is defined
- Inserting a queue name into the service's directory
- Deleting a queue name from the service's directory

It also contains initialization and termination functions.

An installable service is provided by one or more service components. Each component can perform some or all of the functions that are defined for that service. The component is also responsible for managing any underlying resources or software that it needs to implement the service. Configuration files provide a standard way of loading the component and determining the addresses of the functional routines that it provides.

Services and components are related as follows:

- A service is defined to a queue manager by stanzas in a configuration file.
- Each service is supported by supplied code in the queue manager. Users cannot change this code and therefore cannot create their own services.
- Each service is implemented by one or more components; these can be supplied with the product or user-written. Multiple components for a service can be invoked, each supporting different facilities within the service.
- Entry points connect the service components to the supporting code in the queue manager.

Entry-points

Each service component is represented by a list of the entry-point addresses of the routines that support a particular installable service. The installable service defines the function to be performed by each routine. The ordering of the service components when they are configured defines the order in which entry-points are called in an attempt to satisfy a request for the service. In the supplied header file `cmqzc.h`, the supplied entry points to each service have an MQZID_ prefix.

Return codes

Service components provide return codes to the queue manager to report on a variety of conditions. They report the success or failure of the operation, and indicate whether the queue manager is to proceed to the next service component. A separate *Continuation* parameter carries this indication.

Component data

A single service component might require data to be shared between its various functions. Installable services provide an optional data area to be passed on each invocation of a particular service component. This data area is for the exclusive use of the service component. It is shared by all the invocations of a given function, even if they are made from different address spaces or processes. It is guaranteed to be

addressable from the service component whenever it is called. You must declare the size of this area in the *ServiceComponent* stanza.

IBM i *Initialization on IBM i*

When the component initialization routine is invoked, it must call the queue manager MQZEP function for each entry-point supported by the component. MQZEP defines an entry-point to the service. All the undefined exit points are assumed to be NULL.

Primary initialization
A component is always invoked with this option once, before it is invoked in any other way.

Secondary initialization
A component can be invoked with this option on certain platforms. For example, it can be invoked once for each operating system process, thread, or task by which the service is accessed.

If secondary initialization is used:

- The component can be invoked more than once for secondary initialization. For each such call, a matching call for secondary termination is issued when the service is no longer needed.

  For authorization services this is the MQZ_TERM_AUTHORITY call.
- The entry points must be re-specified (by calling MQZEP) each time the component is called for primary and secondary initialization.
- Only one copy of component data is used for the component; there is not a different copy for each secondary initialization.
- The component is not invoked for any other calls to the service (from the operating system process, thread, or task, as appropriate) before secondary initialization has been carried out.
- The component must set the **Version** parameter to the same value for primary and secondary initialization.

Primary termination
The component is always started with this option once, when it is no longer required. No further calls are made to this component.

Secondary termination
The component is started with this option, if it has been started for secondary initialization.

IBM i *Configuring services and components on IBM i*

You configure service components using the queue manager configuration files.

Procedure

1. Add stanzas to the queue manager configuration file, `qm.ini`, to define the service to the queue manager and specify the location of the module:
   - Each service used must have a `Service` stanza, which defines the service to the queue manager. For more information, see Service stanza of the qm.ini file.
   - For each component within a service, there must be a `ServiceComponent` stanza. This stanza identifies the name and path of the module containing the code for that component. For more information, see ServiceComponent stanza of the qm.ini file.

   The authorization service component, known as the Object Authority Manager (OAM), is supplied with the product. When you create a queue manager, the queue manager configuration file is automatically updated to include the appropriate stanzas for the authorization service and for the default component (the OAM). For the other components, you must configure the queue manager configuration file manually.

   The code for each service component is loaded into the queue manager when the queue manager is started, using dynamic binding, where this is supported on the platform.
2.

IBM i *Creating your own service component on IBM i*

Use this information to learn how to create a service component on IBM MQ for IBM i.

To create your own service component:

- Ensure that the header file `cmqzc.h` is included in your program.
- Create the shared library by compiling the program and linking it with the shared libraries `libmqm*` and `libmqmzf*`.

  Note: Because the agent can run in a threaded environment, you must build the OAM to run in a threaded environment. This includes using the threaded versions of `libmqm` and `libmqmzf`.
- Add stanzas to the queue manager configuration file to define the service to the queue manager and to specify the location of the module.
- Stop and restart the queue manager to activate the component.

IBM i *Authorization service on IBM i*

The authorization service is an installable service that enables queue managers to invoke authorization facilities, for example, checking that a user ID has authority to open a queue.

This service is a component of the IBM MQ security enabling interface (SEI), which is part of the IBM MQ framework. The following subjects are discussed:

- "Object authority manager (OAM)" on page 916
- "Defining the service to the operating system" on page 916
- "Configuring authorization service stanzas" on page 917
- "Authorization service interface on IBM i" on page 917

Object authority manager (OAM)

The authorization service component supplied with the IBM MQ products is called the object authority manager (OAM). By default, the OAM is active and works with the following control commands:

- **WRKMQMAUT** work with authority
- **WRKMQMAUTD** work with authority data
- **DSPMQMAUT** display object authority
- **GRTMQMAUT** grant object authority
- **RVKMQMAUT** revoke object authority
- **RFRMQMAUT** refresh security

The syntax of these commands and how to use them are described in the CL command help. The OAM works with the *entity* of a principal or group.

When an MQI request is made or a command is issued, the OAM checks the authorization of the entity associated with the operation to see whether it can do the following actions:

- Perform the requested operation.
- Access the specified queue manager resources.

The authorization service enables you to augment or replace the authority checking provided for queue managers by writing your own authorization service component.

Defining the service to the operating system

The authorization service stanzas in the queue manager configuration file `qm.ini` define the authorization service to the queue manager. See "Configuring services and components on IBM i" on page 915 for information about the types of stanza.

Configuring authorization service stanzas

On IBM MQ for IBM i:

Principal
Is an IBM i system user profile.

Group
Is an IBM i system group profile.

Authorizations can be granted or revoked at the group level only. A request to grant or revoke a user's authority updates the primary group for that user.

Each queue manager has its own queue manager configuration file. For example, the default path and file name of the queue manager configuration file for queue manager QMNAME is `/QIBM/UserData/mqm/qmgrs/QMNAME/qm.ini`.

The *Service* stanza and the *ServiceComponent* stanza for the default authorization component are added to `qm.ini` automatically, but can be overridden by `WRKENVVAR`. Any other *ServiceComponent* stanzas must be added manually.

For example, the following stanzas in the queue manager configuration file define two authorization service components:

```
Service:
    Name=AuthorizationService
    EntryPoints=7

ServiceComponent:
    Service=AuthorizationService
    Name=MQ.UNIX.authorization.service
    Module=QMQM/AMQZFU
    ComponentDataSize=0

ServiceComponent:
    Service=AuthorizationService
    Name=user.defined.authorization.service
    Module=LIBRARY/SERVICE PROGRAM NAME
    ComponentDataSize=96
```

Figure 100. Authorization service stanzas in qm.ini on IBM i

The first service component stanza `MQ.UNIX.authorization.service` defines the default authorization service component, the OAM. If you remove this stanza and restart the queue manager, the OAM is disabled and no authorization checks are made.

IBM i *Authorization service interface on IBM i*

The authorization service interface provides several entry points for use by the queue manager.

MQZ_AUTHENTICATE_USER
Authenticates a user ID and password, and can set identity context fields.

MQZ_CHECK_AUTHORITY
Checks whether an entity has authority to perform one or more operations on a specified object.

MQZ_COPY_ALL_AUTHORITY
Copies all the current authorizations that exist for a referenced object to another object.

MQZ_DELETE_AUTHORITY
Deletes all authorizations associated with a specified object.

MQZ_ENUMERATE_AUTHORITY_DATA
Retrieves all the authority data that matches the selection criteria specified.

MQZ_FREE_USER
Frees associated allocated resources.

MQZ_GET_AUTHORITY
Gets the authority that an entity has to access a specified object.

MQZ_GET_EXPLICIT_AUTHORITY
Gets either the authority that a named group has to access a specified object (but without the additional authority of the **nobody** group) or the authority that the primary group of the named principal has to access a specified object.

MQZ_INIT_AUTHORITY
Initializes authorization service component.

MQZ_INQUIRE
Queries the supported functionality of the authorization service.

MQZ_REFRESH_CACHE
Refresh all authorizations.

MQZ_SET_AUTHORITY
Sets the authority that an entity has to a specified object.

MQZ_TERM_AUTHORITY
Terminates authorization service component.

These entry points support the use of the Windows Security Identifier (NT SID).

These names are defined as **typedef** s, in the header file `cmqzc.h`, which can be used to prototype the component functions.

The initialization function (**MQZ_INIT_AUTHORITY**) must be the main entry point for the component. The other functions are invoked through the entry point address that the initialization function has added into the component entry point vector.

See "Creating your own service component on IBM i" on page 916 for more information.

Multi Writing and compiling API exits on Multiplatforms

API exits let you write code that changes the behavior of IBM MQ API calls, such as MQPUT and MQGET, and then insert that code immediately before or immediately after those calls.

Note: z/OS Not supported on IBM MQ for z/OS.

Why use API exits?

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:

Security
For security, you can provide authentication, checking that applications are authorized to access a queue or queue manager. You can also police applications' use of the API, authenticating the individual API calls, or even the parameters they use.

Flexibility
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.

Monitoring use of a queue or queue manager
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.

What happens when an API exit runs?

Once you have written an exit program and identified it to IBM MQ, the queue manager automatically invokes your exit code at the registered points.

The API exit routines to run are identified in stanzas on Multiplatforms. This topic covers the stanzas in the configuration files `mqs.ini` and `qm.ini`.

The definition of the routines can occur in three places:

1. ApiExitCommon, in the mqs.ini file, identifies routines, for the whole of IBM MQ, applied when queue managers start. These can be overridden by routines defined for individual queue managers (see item "3" on page 919 in this list).
2. ApiExitTemplate, in the mqs.ini file, identifies routines, for the whole of IBM MQ, copied to the ApiExitLocal set (see item "3" on page 919 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 "Configuring API exits" on page 923 determines the order in which the routines defined in the stanzas run.

Using API exits across multiple installations of IBM MQ

Ensure that the API exits written for the earlier version of IBM MQ are used to work with all versions because the changes made to exits in IBM WebSphere MQ 7.1 might not work with an earlier version. For more information about the changes made to exits, see "Writing exits and installable services on AIX, Linux, and Windows" on page 902.

The samples provided for API exits amqsaem and amqsaxe reflect the changes required while writing exits. The client application must ensure that the correct IBM MQ libraries that correspond to the installation of the queue manager with which the application is associated are linked to it prior to the launch of the application.

Multi ***Writing API exits***

You can write exits for every API call using the C programming language.

Available exits

Exits are available for every API call, as follows:

- MQCB, to reregister a callback for the specified object handle and control activation and changes to the callback
- MQCTL, to perform controlling actions on the object handles opened for a connection
- MQCONN/MQCONNX, to provide a queue manager connection handle for use on subsequent API calls
- MQDISC, to disconnect from a queue manager
- MQBEGIN, to begin a global unit of work (UOW)
- MQBACK, to back out a UOW
- MQCMIT, to commit a UOW
- MQOPEN, to open an IBM MQ resource for subsequent access
- MQCLOSE, to close an IBM MQ resource that had previously been opened for access
- MQGET, to retrieve a message from a queue that has previously been opened for access
- MQPUT1, to place a message on to a queue
- MQPUT, to place a message on to a queue that has previously been opened for access

- MQINQ, to inquire on the attributes of an IBM MQ resource that has previously been opened for access
- MQSET, to set the attributes of a queue that has previously been opened for access
- MQSTAT, to retrieve status information
- MQSUB, to register the applications subscription to a particular topic
- MQSUBRQ, to make a request for a subscription

MQ_CALLBACK_EXIT provides an exit function to perform before and after callback processing. For more information, see Callback - MQ_CALLBACK_EXIT.

Writing API exits

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. The *parameters* control the function of the exit, primarily providing communication between the exit and the external control blocks MQAXP (the API exit parameter structure) and MQAXC (the API exit context structure). *context* describes the context in which the API exit was called, and *ApiCallParameters* represent the parameters to the MQI call.

To help you write your API exit, a sample exit, amqsaxe0.c, is provided; this exit generates trace entries to a file that you specify. You can use this sample as your starting point when writing exits. For more information about using the sample exit, see "The API exit sample program" on page 1035.

For more information about the API exit calls, external control blocks, and associated topics, see API exit reference.

For general information on how to write, compile and configure an exit, see "Writing exits and installable services on AIX, Linux, and Windows" on page 902.

Using message handles in API exits

You can control which message properties an API exit has access to. Properties are associated with an ExitMsgHandle. Properties set in a put exit are set on the message being put, but properties retrieved in a get exit are not returned to the application.

When you register an MQ_INIT_EXIT exit function using the MQXEP MQI call with **Function** set to MQXF_INIT and **ExitReason** set to MQXR_CONNECTION, you pass in an MQXEPO structure as the **ExitOpts** parameter. The MQXEPO structure contains the ExitProperties field, which specifies the set of properties to be made available to the exit. It is specified as a character string representing the prefix of the properties, which corresponds to an MQRFH2 folder name.

Each API exit receives an MQAXP structure, containing an ExitMsgHandle field. This field is set to a value generated by IBM MQ and is specific to a connection. The handle is therefore unchanged between API exits of the same or different types on the same connection.

In an MQ_PUT_EXIT or MQ_PUT1_EXIT with an **ExitReason** of MQXR_BEFORE, that is, an API exit performed before putting a message, any properties (other than message descriptor properties) associated with the ExitMsgHandle when the exit completes are set on the message being put. To prevent this happening, set ExitMsgHandle to MQHM_NONE. You can also supply a different message handle.

In an MQ_GET_EXIT and MQ_CALLBACK_EXIT, the ExitMsgHandle is cleared of properties and populated with the properties specified in the ExitProperties field when the MQ_INIT_EXIT was registered, other than message descriptor properties. These properties are not made available to the getting application. If the getting application specified a message handle in the MQGMO (Get message options) field, then any properties associated with that handle, including message descriptor properties, are available to the API exit. To prevent the ExitMsgHandle being populated with properties, set it to MQHM_NONE.

A sample program, amqsaem0.c, is provided to illustrate the use of message handles in API exits.

Related reference

User exits, API exits, and installable services reference

Multi *Compiling API exits*

After 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 1035. For platforms other than Windows systems, you can find the sample API exit code in *MQ_INSTALLATION_PATH*/samp and the compiled and linked shared library in *MQ_INSTALLATION_PATH*/samp/bin.

Windows For Windows systems, you can find the sample API exit code in *MQ_INSTALLATION_PATH* \Tools\c\Samples. *MQ_INSTALLATION_PATH* represents the directory in which IBM MQ was installed.

Note: Guidance on programming 64 bit applications is listed in Coding standards on 64-bit platforms.

For Multicast clients, API exits and data-conversion exits need to be able to run on the client-side because some messages might not go through the queue manager. The following libraries are part of the client packages as well as the server packages:

Table 141. Libraries that are in the client and server packages

Operating system	Libraries
AIX AIX	32 bit & 64 bit: libmqm.a & libmqm_r.a
IBM i IBM i	LIBMQM & LIBMQM_R
Linux Linux	32 bit & 64 bit: libmqm.so & libmqm_r.so
Windows Windows	32 bit & 64 bit: mqm.dll & mqm.pdb

Linux AIX *Compiling API exits on AIX and Linux systems*

Examples of how to Compile API exits on AIX and Linux systems.

On all platforms, the entry point to the module is MQStart.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

On AIX

AIX

Compile the API exit source code by issuing one of the following commands:

32 bit applications

Non-threaded

```
cc -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits/amqsaxe \
amqsaxe0.c -I MQ_INSTALLATION_PATH/inc
```

Threaded

```
xlc_r -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits/amqsaxe_r \
amqsaxe0.c -I MQ_INSTALLATION_PATH/inc
```

64 bit applications

Non-threaded

```
cc -q64 -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits64/amqsaxe \
amqsaxe0.c -I MQ_INSTALLATION_PATH/inc
```

Threaded

```
xlc_r -q64 -e MQStart -bE:amqsaxe.exp -bM:SRE -o /var/mqm/exits64/amqsaxe_r \
amqsaxe0.c -I MQ_INSTALLATION_PATH/inc
```

On Linux

Linux

Compile the API exit source code by issuing one of the following commands:

31 bit applications

Non-threaded

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/amqsaxe amqsaxe0.c \
    -I MQ_INSTALLATION_PATH/inc
```

Threaded

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/amqsaxe_r amqsaxe0.c \
    -I MQ_INSTALLATION_PATH/inc
```

32 bit applications

Non-threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/amqsaxe amqsaxe0.c \
    -I MQ_INSTALLATION_PATH/inc
```

Threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/amqsaxe_r amqsaxe0.c \
    -I MQ_INSTALLATION_PATH/inc
```

64 bit applications

Non-threaded

```
gcc -m64 -shared -fPIC -o /var/mqm/exits64/amqsaxe amqsaxe0.c \
    -I MQ_INSTALLATION_PATH/inc
```

Threaded

```
gcc -m64 -shared -fPIC -o /var/mqm/exits64/amqsaxe_r amqsaxe0.c \
    -I MQ_INSTALLATION_PATH/inc
```

Windows *Compiling API exits on Windows systems*

Compile and link the sample API exit program, `amqsaxe0.c`, on Windows

A manifest file is an optional XML document containing the version, or any other, information that can be embedded in a compiled application or DLL.

If you have no such document, omit the -manifest *manifest.file* parameter in the **mt** command.

Adapt the commands in the examples in Figure 101 on page 923 or Figure 102 on page 923 to compile and link `amqsaxe0.c` on Windows. The commands work with Microsoft Visual Studio 2008, 2010, or 2012. The examples assume that the `C:\Program Files\IBM\MQ\tools\c\samples` directory is the current directory.

32 bit

```
cl /c /nologo /MD /Foamqsaxe0.obj amqsaxe0.c

link /nologo /dll /def:amqsaxe.def



amqsaxe0.obj \
   /manifest /out:amqsaxe.dll

mt -nologo -manifest amqsaxe.dll.manifest \
   -outputresource:amqsaxe.dll;2
```

Figure 101. Compile and link `amqsaxe0.c` on 32 bit Windows

64 bit

```
cl /c /nologo /MD /Foamqsaxe0.obj amqsaxe0.c

link /nologo /dll /def:amqsaxe.def \
     /libpath:..\..\lib64 \



amqsaxe0.obj /manifest /out:amqsaxe.dll

mt -nologo -manifest amqsaxe.dll.manifest \
   -outputresource:amqsaxe.dll;2
```

Figure 102. Compile and link `amqsaxe0.c` on 64 bit Windows

Related concepts

"The API exit sample program" on page 1035
The sample API exit generates an MQI trace to a user-specified file with a prefix defined in the **MQAPI_TRACE_LOGFILE** environment variable.

IBM i *Compliling API exits on IBM i*

Compiling API exits on IBM i.

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.

Configuring API exits

You configure IBM MQ to enable API exits by changing the configuration information.

To change the configuration information, you must change the stanzas that define the exit routines and the sequence in which they run. This information can be changed in the following ways:

- Windows Linux Using the IBM MQ Explorer on Windows and Linux (x86 and x86-64 platforms).
- Windows Using the **amqmdain** command on Windows.
- Multi Using the `mqs.ini` and `qm.ini` files directly on Multiplatforms.

  The `mqs.ini` file contains information relevant to all the queue managers on a particular node. You can find it in the following locations:

  – Linux AIX In the `/var/mqm` directory on AIX and Linux.

  – Windows In the WorkPath specified in the `HKLM\SOFTWARE\IBM\WebSphere MQ` key on Windows systems.

  – IBM i In the `/QIBM/UserData/mqm` directory on IBM i.

  The `qm.ini` file contains information relevant to a specific queue manager. There is one queue manager configuration file for each queue manager, held in the root of the directory tree occupied by the queue manager. For example, the path and the name for a configuration file for a queue manager called QMNAME is:

  IBM i On IBM i systems:

  ```
  /QIBM/UserData/mqm/qmgrs/QMNAME/qm.ini
  ```

  Linux AIX On AIX and Linux systems:

  ```
  /var/mqm/qmgrs/QMNAME/qm.ini
  ```

  Windows On Windows systems:

  ```
  C:\ProgramData\IBM\MQ\qmgrs\QMNAME\qm.ini
  ```

  Before editing a configuration file, back it up so that you have a copy you can revert to if the need arises.

  You can edit configuration files in either of the following ways:

  – Automatically, using commands that change the configuration of queue managers on the node.

  – Manually, using a standard text editor.

  If you set an incorrect value on a configuration file attribute, the value is ignored and an operator message is issued to indicate the problem. The effect is the same as missing out the attribute entirely.

Stanzas to configure

The stanzas that must be changed are the following:

ApiExitCommon

Defined in `mqs.ini` and in the IBM MQ Explorer on the IBM MQ properties page, under Exits.

When any queue manager starts, the attributes in this stanza are read, and then overridden by the API exits defined in `qm.ini`.

ApiExitTemplate

Defined in `mqs.ini` and in the IBM MQ Explorer on the IBM MQ properties page, under Exits.

When any queue manager is created, the attributes in this stanza are copied into the newly created qm.ini file under the ApiExitLocal stanza.

ApiExitLocal

Defined in `qm.ini` and in the IBM MQ Explorer on the queue manager properties page, under Exits.

When the queue manager starts, API exits defined here override the defaults defined in `mqs.ini`.

Attributes for the stanzas

- Name the API exit using the following attribute:

  Name=ApiExit_name
  The descriptive name of the API exit passed to it in the ExitInfoName field of the MQAXP structure.

  This name must be unique, no longer than 48 characters, and contain only valid characters for the names of IBM MQ objects (for example, queue names).

- Identify the module and entry point of the API exit code to run using the following attributes:

  Function=function_name
  The name of the function entry point into the module containing the API exit code. This entry point is the MQ_INIT_EXIT function.

  The length of this field is limited to MQ_EXIT_NAME_LENGTH.

  Module=module_name
  The module containing the API exit code.

  If this field contains the full path name of the module it is used as is.

  If this field contains just the module name, the module is located using the `ExitsDefaultPath` attribute in the `ExitPath` in qm.ini.

  On platforms that support separate threaded libraries, you must provide both a non-threaded and a threaded version of the API exit module. The threaded version must have an `_r` suffix. The threaded version of the IBM MQ application stub implicitly appends `_r` to the given module name before it is loaded.

  The length of this field is limited to the maximum path length the platform supports.

- Optionally pass data with the exit using the following attribute:

  Data=data_name
  Data to be passed to the API exit in the ExitData field of the MQAXP structure.

  If you include this attribute, leading and trailing blanks are removed, the remaining string is truncated to 32 characters, and the result is passed to the exit. If you omit this attribute, the default value of 32 blanks is passed to the exit.

  The maximum length of this field is 32 characters.

- Identify the sequence of this exit in relation to other exits using the following attribute:

  Sequence=sequence_number
  The sequence in which this API exit is called relative to other API exits. An exit with a low sequence number is called before an exit with a higher sequence number. There is no need for the sequence numbering of exits to be contiguous. A sequence of 1, 2, 3 has the same result as a sequence of 7, 42, 1096. If two exits have the same sequence number, the queue manager decides which one to call first. You can tell which was called after the event by putting the time or a marker in ExitChainArea indicated by the ExitChainAreaPtr in MQAXP or by writing your own log file.

  This attribute is an unsigned numeric value.

Sample stanzas

The sample mqs.ini file contains the following stanzas:

ApiExitTemplate
This stanza defines an exit with the descriptive name `OurPayrollQueueAuditor`, module name `auditor`, and sequence number 2. A data value of 123 is passed to the exit.

ApiExitCommon
This stanza defines an exit with the descriptive name `MQPoliceman`, module name `tmqp`, and sequence number 1. The data passed is an instruction (`CheckEverything`).

```
mqs.ini

   ApiExitTemplate:
      Name=OurPayrollQueueAuditor
      Sequence=2
      Function=EntryPoint
      Module=/usr/ABC/auditor
      Data=123
   ApiExitCommon:
      Name=MQPoliceman
      Sequence=1
      Function=EntryPoint
      Module=/usr/MQPolice/tmqp
      Data=CheckEverything
```

The following sample qm.ini file contains an ApiExitLocal definition of an exit with the descriptive name `ClientApplicationAPIchecker`, module name `ClientAppChecker`, and sequence number 3.

```
qm.ini

   ApiExitLocal:
      Name=ClientApplicationAPIchecker
      Sequence=3
      Function=EntryPoint
      Module=/usr/Dev/ClientAppChecker
      Data=9.20.176.20
```

Channel-exit programs for messaging channels

This collection of topics contains information about IBM MQ channel-exit programs for messaging channels.

Message channel agents (MCAs) can also call data-conversion exits. For more information about writing data-conversion exits, see “Writing data-conversion exits” on page 947.

Some of this information also applies to exits on MQI channels, which connect IBM MQ MQI clients to queue managers. For more information, see Channel-exit programs for MQI channels.

Channel-exit programs are called at defined places in the processing carried out by MCA programs.

Some of these user-exit programs work in complementary pairs. For example, if a user-exit program is called by the sending MCA to encrypt the messages for transmission, the complementary process must be functioning at the receiving end to reverse the process.

Table 142 on page 926 shows the types of channel exit that are available for each channel type.

Table 142. Channel exits available for each channel type

Channel Type	Message exit	Message-retry exit	Receive exit	Security exit	Send exit	Auto-definition exit
Sender channel	Yes		Yes	Yes	Yes	
Server channel	Yes		Yes	Yes	Yes	
Cluster-sender channel	Yes		Yes	Yes	Yes	Yes

Table 142. Channel exits available for each channel type (continued)						
Channel Type	**Message exit**	**Message-retry exit**	**Receive exit**	**Security exit**	**Send exit**	**Auto-definition exit**
Receiver channel	Yes	Yes	Yes	Yes	Yes	Yes
Requester channel	Yes	Yes	Yes	Yes	Yes	
Cluster-receiver channel	Yes	Yes	Yes	Yes	Yes	Yes
Client-connection channel			Yes	Yes	Yes	
Server-connection channel			Yes	Yes	Yes	Yes

Notes: z/OS

1. On z/OS, the auto-definition exit applies to cluster-sender and cluster-receiver channels only.

If you are going to run channel exits on a client, you cannot use the MQSERVER environment variable. Instead, create and reference a client channel definition table (CCDT) as described in Client channel definition table.

Processing overview

An overview of how MCAs use channel-exit programs.

On startup, the MCAs exchange a startup dialog to synchronize processing. Then they switch to a data exchange that includes the security exits. These exits must end successfully for the startup phase to complete and to allow messages to be transferred.

The security check phase is a loop, as shown in Figure 103 on page 927.

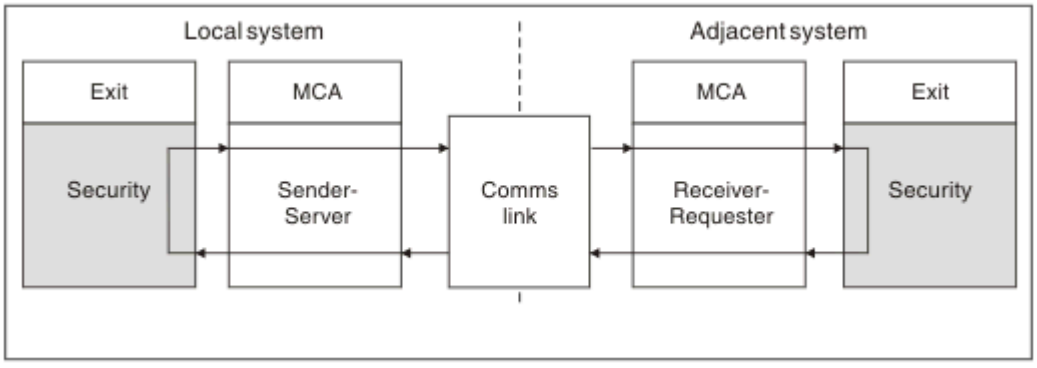


Figure 103. Security exit loop

During the message transfer phase, the sending MCA gets messages from a transmission queue, calls the message exit, calls the send exit, and then sends the message to the receiving MCA, as shown in Figure 104 on page 928.

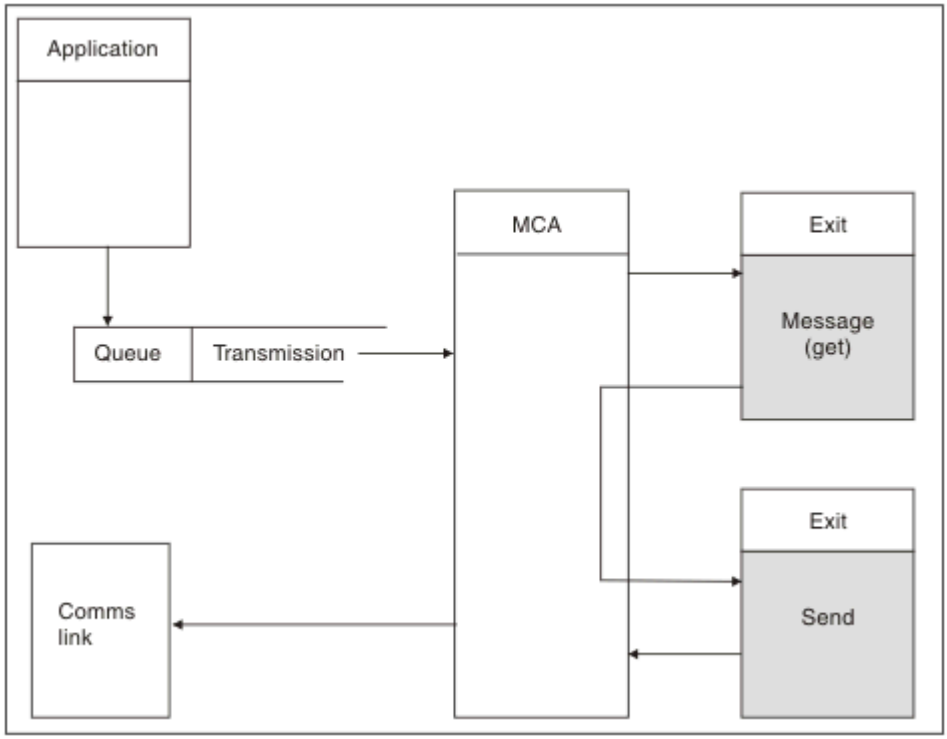


Figure 104. Example of a send exit at the sender end of message channel

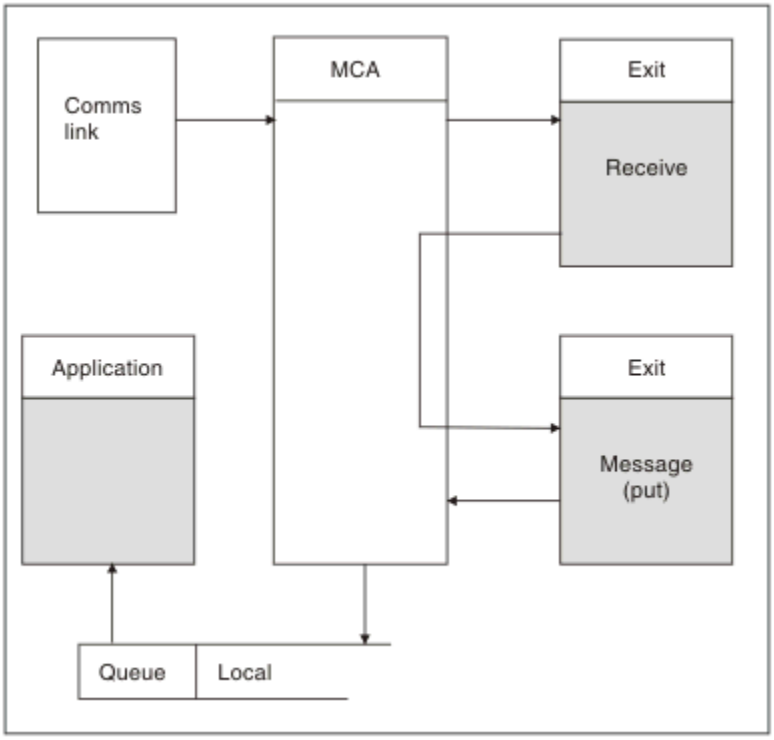


Figure 105. Example of a receive exit at the receiver end of message channel

The receiving MCA receives a message from the communications link, calls the receive exit, calls the message exit, and then puts the message on the local queue, as shown in Figure 105 on page 928. (The receive exit can be called more than once before the message exit is called.)

Writing channel-exit programs

You can use the following information to help you write channel-exit programs.

User exits and channel-exit programs can use all MQI calls, except as noted in the sections that follow. For MQ V7 and later, the MQCXP structure version 7 and higher contains the connection handle hConn, which can be used instead of issuing MQCONN. For earlier versions, to obtain the connection handle, an MQCONN must be issued, even though an MQRC_ALREADY_CONNECTED warning is returned because the channel itself is connected to the queue manager.

Note that the channel exit must be threadsafe.

For exits on client-connection channels, the queue manager to which the exit tries to connect depends on how the exit was linked. If the exit was linked with MQM.LIB (or QMQM/LIBMQM on IBM i) and you do not specify a queue manager name on the MQCONN call, the exit tries to connect to the default queue manager on your system. If the exit was linked with MQM.LIB (or QMQM/LIBMQM on IBM i) and you specify the name of the queue manager that was passed to the exit through the QMgrName field of MQCD, the exit tries to connect to that queue manager. If the exit was linked with MQIC.LIB or any other library, the MQCONN call fails whether you specify a queue manager name or not.

You should avoid altering the state of the transaction associated with the passed hConn in a channel exit; you must not use the MQCMIT, MQBACK or MQDISC verbs with the channel hConn, and you cannot use the MQBEGIN verb specifying the channel hConn.

If MQCONNX is used specifying MQCNO_HANDLE_SHARE_BLOCK or MQCNO_HANDLE_SHARE_NO_BLOCK to create a new IBM MQ connection, then it is your responsibility to ensure that the connection is correctly managed and disconnects from the queue manager correctly. For example, a channel exit that creates a new connection to the queue manager on every invocation without disconnecting, results in connection handles building up and an increase in the number of agent threads.

An exit runs in the same thread as the MCA itself and uses the same connection handle. So, it runs inside the same UOW as the MCA and any calls made under sync point are committed or backed out by the channel at the end of the batch.

Therefore, a channel message exit could send notification messages that are only committed to that queue when the batch containing the original message is committed. So, it is possible to issue sync point MQI calls from a channel message exit.

A channel exit can change fields in the MQCD. However, these changes are not acted on, except in the circumstances listed. If a channel exit program changes a field in the MQCD data structure, the new value is ignored by the IBM MQ channel process. However, the new value remains in the MQCD and is passed to any remaining exits in an exit chain and to any conversation sharing the channel instance. For more information, see Changing MQCD fields in a channel exit

Also, for programs written in C, non-reentrant C library function must not be used in a channel-exit program.

If you use multiple channel exit libraries simultaneously, problems can arise on some UNIX and Linux platforms if the code for two different exits contains identically named functions. When a channel exit is loaded, the dynamic loader resolves function names in the exit library to the addresses where the library is loaded. If two exit libraries define separate functions which happen to have identical names, this resolution process might incorrectly resolve the function names of one library to use the functions of another. If this problem occurs, specify to the linker that it must only export the required exit and MQStart functions, as these functions are unaffected. Other functions must be given local visibility so that they are not used by functions outside their own exit library. Consult the documentation for the linker for more information.

All exits are called with a channel exit parameter structure (MQCXP), a channel definition structure (MQCD), a prepared data buffer, data length parameter, and buffer length parameter. The buffer length must not be exceeded:

- For message exits, you must allow for the largest message required to be sent across the channel, plus the length of the MQXQH structure.
- For send and receive exits, the largest buffer you must allow for is as follows:

LU 6.2
32 KB

TCP:

IBM i IBM i 16 KB

IBM i Others 32 KB

Note: The maximum usable length might be 2 bytes less than this length. Check the value returned in **MaxSegmentLength** for details. For more information about **MaxSegmentLength**, see MaxSegmentLength.

NetBIOS:
64 KB

SPX:
64 KB

Note: Receive exits on sender channels and sender exits on receiver channels use 2 KB buffers for TCP.

- For security exits, the distributed queuing facility allocates a buffer of 4000 bytes.

It is permissible for the exit to return an alternative buffer, together with the relevant parameters. See “Channel-exit programs for messaging channels” on page 926 for call details.

z/OS *Writing channel exit programs on z/OS*

You can use the following information to help you write and compile channel-exit programs for z/OS.

The exits are started as if by a 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

The link-edited modules must be placed in the data set specified by the CSQXLIB DD statement of the channel initiator address space procedure; the names of the load modules are specified as the exit names in the channel definition.

When writing channel exits for z/OS, the following rules apply:

- Exits must be written in assembler or C; if C is used, it must conform to the C systems programming environment for system exits, described in the z/OS C/C++ Programming Guide.
- Exits are loaded from the non-authorized libraries defined by a CSQXLIB DD statement. Providing CSQXLIB has DISP=SHR, exits can be updated while the channel initiator is running. The new version is used when the channel is restarted.
- Exits must be reentrant, and capable of running anywhere in virtual storage.
- Exits must reset the environment, on return, to that at entry.
- Exits must free any storage obtained, or ensure that it is freed by a subsequent exit invocation.

  For storage that is to persist between invocations, use the z/OS STORAGE service, or the 4kmalc library function for System Programming C.

  For more information about this function, see 4kmalc() -- Allocate Page-Aligned Storage.
- All IBM MQ MQI calls except MQCMIT or CSQBCMT and MQBACK or CSQBBAK can be used. They must be contained after MQCONN (with a blank queue manager name). If these calls are used, the exit must be link-edited with the stub CSQXSTUB.

  The exception to this rule is that security channel exits can issue commit and backout MQI calls. To issue such calls, code the verbs CSQXCMT and CSQXBAK in place of MQCMIT or CSQBCMT and MQBACK or CSQBBAK.

- All exits that use stub CSQXSTUB from IBM WebSphere MQ 7.0 or later must be link-edited in a CSQXLIB load library with format PDS-E.
- Exits must not use any system services that cause a wait, because using system services would severely affect the handling of some or all the other channels. Many channels are run under a single TCB typically. If you do something in an exit that causes a wait and you do not use MQXWAIT, it causes all these channels to wait. Causing channels to wait does not give any functional problems, but might have an adverse effect on performance. Most SVCs involve waits, so you must avoid them, except for the following SVCs:
  - GETMAIN/FREEMAIN/STORAGE
  - LOAD/DELETE

  In general, therefore, avoid SVCs, PCs, and I/O. Instead, use the MQXWAIT call.
- Exits do not issue ESTAEs or SPIEs, apart from in any subtasks they attach, because their error handling might interfere with the error handling performed by IBM MQ. This means that IBM MQ might not be able to recover from an error, or that your exit program might not receive all the error information.
- The MQXWAIT call (see MQXWAIT) provides a wait service that waits for I/O and other events; if this service is used, exits must not use the linkage stack.

  For I/O and other facilities that do not provide non-blocking facilities or an ECB to wait on, a separate subtask must be ATTACHed, and its completion waited for by MQXWAIT; because of the processing that this technique incurs, this facility must be used only by the security exit.
- The MQDISC MQI call does not cause an implicit commit to occur within the exit program. A commit of the channel process is performed only when the channel protocol dictates.

The following exit samples are provided with IBM MQ for z/OS:

CSQ4BAX0
This sample is written in assembler, and illustrates the use of MQXWAIT.

CSQ4BCX1 and CSQ4BCX2
These samples are written in C and illustrate how to access the parameters.

CSQ4BCX3 and CSQ4BAX3
These samples are written in C and assembler respectively.

The CSQ4BCX3 sample (which is pre-compiled into the SCSQAUTH LOADLIB, should function with no changes necessary on the exit itself. You can create a LOADLIB (for example, called MY.TEST.LOADLIB) and copy the SCSQAUTH(CSQ4BCX3) member to it.

To set up a security exit on a client connection, carry out the following procedure:

1. Establish a valid OMVS segment for the user ID that the channel initiator uses.

   This allows the IBM MQ for z/OS channel initiator to use TCP/IP with the z/OS UNIX System Services (z/OS UNIX) socket interface, in order to facilitate exit processing. Note that it is unnecessary to define an OMVS segment for the user ID of any connecting client.
2. Ensure that the exit code itself runs only in a program controlled environment.

   This means everything loaded into the CHINIT address space must be loaded from a program controlled library (meaning all libraries in the STEPLIB), and any libraries named on CSQXLIB and

   ```
   ++hlq++.SCSQANLx
   ++hlq++.SCSQMVR1
   ++hlq++.SCSQAUTH
   ```

   To set a load library as program controlled, use a command similar to this example:

   ```
   RALTER PROGRAM * ADDMEM('MY.TEST.LOADLIB'//NOPADCHK)
   ```

   Then you can activate or refresh the program controlled environment by issuing the command:

```
SETROPTS WHEN(PROGRAM) REFRESH
```

3. Add the exit LOADLIB to the CSQXLIB DD (in the CHINIT started procedure), by issuing the following command:

```
ALTER CHANNEL(xxxx) CHLTYPE(SVRCONN)SCYEXIT(CSQ4BCX3)
```

   This activates the exit for the named channel.

4. Your external security manager (ESM) lists any other libraries to be program controlled, but note that none of the ESM or C libraries needs to be under program control.

See IBM MQ for z/OS server connection channel for further information on setting up a security exit using the sample CSQ4BCX3.

CSQ4BCX4
This sample is written in C and demonstrates using the **RemoteProduct** and **RemoteVersion** fields in MQCXP.

Related concepts
“Writing channel exit programs on IBM i” on page 932
You can use the following information to help you write and compile channel-exit programs for IBM i.

“Writing channel-exit programs on AIX, Linux, and Windows” on page 933
You can use the following information to help you write channel-exit programs for AIX, Linux, and Windows systems.

Related reference
IBM MQ for z/OS server connection channel

IBM i *Writing channel exit programs on IBM i*
You can use the following information to help you write and compile channel-exit programs for IBM i.

The exit is a program object written in the ILE C, ILE RPG, or ILE COBOL language. The exit program names and their libraries are named in the channel definition.

Observe the following conditions when creating and compiling an exit program:

- The program must be made thread safe and created with the ILE C, ILE RPG, or ILE COBOL compiler. For ILE RPG you must specify the THREAD(*SERIALIZE) control specification, and for ILE COBOL you must specify SERIALIZE for the THREAD option of the PROCESS statement. The programs must also be bound to the threaded IBM MQ libraries: QMQM/LIBMQM_R in the case of ILE C and ILE RPG, and AMQ0STUB_R in the case of ILE COBOL. For additional information about making RPG or COBOL applications thread safe, refer to the appropriate Programmer's Guide for the language.
- IBM MQ for IBM i requires that the exit programs are enabled for teraspace support. (Teraspace is a form of shared memory introduced in OS/400 V4R4.) For the ILE RPG and COBOL compilers, any programs compiled on OS/400 V4R4 or later are so enabled. For C, the programs must be compiled with the TERASPACE(*YES *TSIFC) options specified on CRTCMOD or CRTBNDC commands.
- An exit returning a pointer to its own buffer space must ensure that the object pointed to exists beyond the time span of the channel-exit program. The pointer cannot be the address of a variable on the program stack, nor of a variable in the program heap. Instead, the pointer must be obtained from the system. An example is a user space created in the user exit. To ensure that any data area allocated by the channel-exit program is still available for the MCA when the program ends, the channel exit must run in the activation group of the caller or a named activation group. Do this by setting the ACTGRP parameter on CRTPGM to a user-defined value or *CALLER. If the program is created in this way, the channel-exit program can allocate dynamic memory and pass a pointer to this memory back to the MCA.

Related concepts
“Writing channel-exit programs on AIX, Linux, and Windows” on page 933

You can use the following information to help you write channel-exit programs for AIX, Linux, and Windows systems.

“Writing channel exit programs on z/OS ” on page 930
You can use the following information to help you write and compile channel-exit programs for z/OS.

ALW *Writing channel-exit programs on AIX, Linux, and Windows*
You can use the following information to help you write channel-exit programs for AIX, Linux, and Windows systems.

Follow the instructions outlined in “Writing exits and installable services on AIX, Linux, and Windows” on page 902. Use the following channel exit specific information, where appropriate:

The exit must be written in C, and is a DLL on Windows.

Define a dummy MQStart() routine in the exit and specify `MQStart` as the entry point in the library. Figure 106 on page 933 shows how to set up an entry to your program:

```
#include <cmqec.h>

void MQStart() {;} /* dummy entry point - for consistency only */
void MQENTRY ChannelExit ( PMQCXP  pChannelExitParms,
                           PMQCD   pChannelDefinition,
                           PMQLONG pDataLength,
                           PMQLONG pAgentBufferLength,
                           PMQVOID pAgentBuffer,
                           PMQLONG pExitBufferLength,
                           PMQPTR  pExitBufferAddr)
{
... Insert code here
}
```

Figure 106. Sample source code for a channel exit

When writing channel exits for Windows using Visual C++, you must write your own DEF file. An example of how is shown in Figure 107 on page 933. For further information on writing channel exit programs, see “Writing channel-exit programs” on page 929.

```
EXPORTS
ChannelExit
```

Figure 107. Sample DEF file for Windows

Related concepts

“Writing channel exit programs on IBM i” on page 932
You can use the following information to help you write and compile channel-exit programs for IBM i.

“Writing channel exit programs on z/OS ” on page 930
You can use the following information to help you write and compile channel-exit programs for z/OS.

Channel security exit programs
You can use security exit programs to verify that the partner at the other end of a channel is genuine. This is known as authentication.

To specify that a channel must use a security exit, specify the exit name in the **SCYEXIT** field of the channel definition.

Note: Authentication can also be achieved with channel authentication records. Channel authentication records provide great flexibility in preventing access to queue managers from certain users and channels, and in mapping remote users to IBM MQ user identifiers. TLS support is also provided by IBM MQ to authenticate your users and to provide encryption and data integrity checks for your data. For more information about TLS, see TLS security protocols in IBM MQ. However, if you still require more sophisticated (or different) forms of security processing, and other types of checks and security context establishment, consider writing security exits.

The Subject and Issuer DN attributes appear in the following channel status attributes:

- SSLPEER (PCF selector MQCACH_SSL_SHORT_PEER_NAME)
- SSLCERTI (PCF selector MQCACH_SSL_CERT_ISSUER_NAME)

These values are returned by channel status commands as well as the data passed to channel security exits listed, as shown:

- MQCD SSLPeerNamePtr
- MQCXP SSLRemCertIssNamePtr

A security exit can be written in C or Java.

Channel security exit programs are called at the following places in the processing cycle of an MCA:

- At MCA initiation and termination.
- Immediately after the initial data negotiation is finished on channel startup. The receiver or server end of the channel can initiate a security message exchange with the remote end by providing a message to be delivered to the security exit at the remote end. It might also decline to do so. The exit program is started again to process any security message received from the remote end.
- Immediately after the initial data negotiation is finished on channel startup. The sender or requester end of the channel processes a security message received from the remote end, or initiates a security exchange when the remote end cannot. The exit program is started again to process all subsequent security messages that might be received.

A requester channel never gets called with MQXR_INIT_SEC. The channel notifies the server that it has a security exit program, and the server then has the opportunity to initiate a security exit. If it does not have one, it informs the requester and a zero length flow is returned to the exit program.

Note: Avoid sending zero-length security messages.

Examples of the data exchanged by security-exit programs are illustrated in figures Figure 108 on page 935 through Figure 111 on page 937. These examples show the sequence of events that occur involving the security exit of the receiver, and the security exit of the sender. Successive rows in the figures represent the passage of time. In some cases, the events at the receiver and sender are not correlated, and therefore can occur at the same time or at different times. In other cases, an event at one exit program results in a complementary event occurring later at the other exit program. For example, in Figure 108 on page 935:

1. The receiver and sender are each invoked with MQXR_INIT, but these invocations are not correlated and can therefore occur at the same time or at different times.
2. The receiver is next invoked with MQXR_INIT_SEC, but returns MQXCC_OK which requires no complementary event at the sender exit.
3. The sender is next invoked with MQXR_INIT_SEC. This is not correlated with the invocation of the receiver with MQXR_INIT_SEC. The sender returns MQXCC_SEND_SEC_MSG, which causes a complementary event at the receiver exit.
4. The receiver is then invoked with MQXR_SEC_MSG, and returns MQXCC_SEND_SEC_MSG, which causes a complementary event at the sender exit.
5. The sender is then invoked with MQXR_SEC_MSG, and returns MQXCC_OK which requires no complementary event at the receiver exit.

Receiver exit	Sender exit
Invoked with MQXR_INIT Responds with MQXCC_OK	Invoked with MQXR_INIT Responds with MQXCC_OK
Invoked with MQXR_INIT_SEC Responds with MQXCC_OK	
	Invoked with MQXR_INIT_SEC Responds with MQXCC_SEND_SEC_MSG
Invoked with MQXR_SEC_MSG Responds with MQXCC_SEND_SEC_MSG	
	Invoked with MQXR_SEC_MSG Responds with MQXCC_OK
Message transfer begins	

Figure 108. Sender-initiated exchange with agreement

Receiver exit	Sender exit
Invoked with MQXR_INIT Responds with MQXCC_OK	Invoked with MQXR_INIT Responds with MQXCC_OK
Invoked with MQXR_INIT_SEC Responds with MQXCC_OK	
	Invoked with MQXR_INIT_SEC Responds with MQXCC_SEND_SEC_MSG
Invoked with MQXR_SEC_MSG Responds with MQXCC_OK	
	Invoked with MQXR_SEC_MSG Responds with MQXCC_SUPPRESS_FUNCTION
Channel closes	
Invoked with MQXR_TERM Responds with MQXCC_OK	Invoked with MQXR_TERM Responds with MQXCC_OK

Figure 109. Sender-initiated exchange with no agreement

Receiver exit	Sender exit
Invoked with MQXR_INIT Responds with MQXCC_OK	Invoked with MQXR_INIT Responds with MQXCC_OK
Invoked with MQXR_INIT_SEC Responds with MQXCC_SEND_SEC_MSG	
	Invoked with MQXR_SEC_MSG Responds with MQXCC_SEND_SEC_MSG
Invoked with MQXR_SEC_MSG Responds with MQXCC_OK	
Message transfer begins	
Invoked with MQXR_TERM Responds with MQXCC_OK	Invoked with MQXR_TERM Responds with MQXCC_OK

Figure 110. Receiver-initiated exchange with agreement

Receiver exit	Sender exit
Invoked with MQXR_INIT Responds with MQXCC_OK	Invoked with MQXR_INIT Responds with MQXCC_OK
Invoked with MQXR_INIT_SEC Responds with MQXCC_SEND_SEC_MSG	
	Invoked with MQXR_SEC_MSG Responds with MQXCC_OK
Invoked with MQXR_SEC_MSG Responds with MQXCC_SUPPRESS_FUNCTION	
Channel closes	

Figure 111. Receiver-initiated exchange with no agreement

The channel security exit program is passed an agent buffer containing the security data, excluding any transmission headers, generated by the security exit. This data can be any suitable data so that either end of the channel is able to perform security validation.

The security exit program at both the sending and receiving end of the message channel can return either of two response codes to any call:

- Security exchange ended with no errors
- Suppress the channel and close down

Note:

1. The channel security exits typically work in pairs. When you define the appropriate channels, make sure that compatible exit programs are named for both ends of the channel.
2. IBM i In IBM i , security exit programs that have been compiled with `Use adopted authority` (USEADPAUT=*YES) can adopt QMQM or QMQMADM authority. Take care that the exit does not use this feature to pose a security risk to your system.
3. On a TLS channel on which the other end of the channel provides a certificate, the security exit receives the Distinguished Name of the subject of this certificate in the MQCD field accessed by SSLPeerNamePtr and the Distinguished Name of the issuer in the MQCXP field accessed by SSLRemCertIssNamePtr. Uses to which this name can be put are:
   - To restrict access over the TLS channel.
   - To change MQCD.MCAUserIdentifier based on the name.

Related concepts

Transport Layer Security (TLS) concepts

Related reference

Channel authentication records

Writing a security exit

You can write a security exit by using the security exit skeleton code.

Figure 112 on page 938 illustrates how to write a security exit.

```
void MQENTRY MQStart() {;}
void MQENTRY EntryPoint (PMQVOID pChannelExitParms,
                         PMQVOID pChannelDefinition,
                         PMQLONG pDataLength,
                         PMQLONG pAgentBufferLength,
                         PMQVOID pAgentBuffer,
                         PMQLONG pExitBufferLength,
                         PMQPTR  pExitBufferAddr)
{
  PMQCXP pParms = (PMQCXP)pChannelExitParms;
  PMQCD  pChDef = (PMQCD)pChannelDefinition;
  /* TODO: Add Security Exit Code Here */

}
```

Figure 112. Security exit skeleton code

The standard IBM MQ Entry Point MQStart must exist, but is not required to perform any function. The name of the function (EntryPoint in this example) can be changed, but the function must be exported when the library is compiled and linked. As in the previous example, the pointers pChannelExitParms must be cast to PMQCXP and pChannelDefinition must be cast to PMQCD. For general information about calling channel exits and the use of parameters, see MQ_CHANNEL_EXIT. These parameters are used in a security exit as follows:

PMQVOID pChannelExitParms
input/output

Pointer to MQCXP structure - cast to PMQCXP to access fields. This structure is used to communicate between the Exit and MCA. The following fields in the MQCXP are of particular interest for Security Exits:

ExitReason
Tells the Security Exit the current state in the security exchange and is used when deciding what action to take.

ExitResponse
The response to the MCA which dictates the next stage in the security exchange.

ExitResponse2
Extra control flags to govern how the MCA interprets the response of the Security Exit.

ExitUserArea
16 bytes (maximum) of storage which can be used by the Security Exit to maintain state between calls.

ExitData
Contains the data specified in the SCYDATA field of the channel definition (32 bytes padded to the right with blanks).

PMQVOID pChannelDefinition
input/output

Pointer to MQCD structure - cast to PMQCD to access fields. This parameter contains the definition of the channel. The following fields in the MQCD are of particular interest for Security Exits:

ChannelName
The channel name (20 bytes padded to the right with blanks).

ChannelType
A code defining the channel type.

MCA User Identifier
This group of three fields is initialized to the value of the MCAUSER field specified in the channel definition. Any user identifier specified by the Security Exit in these fields is used for access control (not applicable to SDR, SVR, CLNTCONN, or CLUSSDR channels).

MCAUserIdentifier
First 12 bytes of identifier padded to the right with blanks.

LongMCAUserIdPtr
Pointer to a buffer containing the full length identifier (not guaranteed null terminated) takes priority over MCAUserIdentifier.

LongMCAUserIdLength
Length of string pointed to by LongMCAUserIdPtr - must be set if LongMCAUserIdPtr is set.

Remote User Identifier
Only applies to CLNTCONN/SVRCONN channel pairs. If no CLNTCONN Security Exit is defined then these three fields are initialized by the client MCA, so they might contain a user identifier from the environment of the client which can be used by a SVRCONN Security Exit for authentication and when specifying the MCA User Identifier. If a CLNTCONN Security Exit is defined then these fields are not initialized and can be set by the CLNTCONN Security Exit, or security messages can be used to pass a user identifier from Client to Server.

RemoteUserIdentifier
First 12 Bytes of identifier padded to the right with blanks.

LongRemoteUserIdPtr
Pointer to a buffer containing the full length identifier (not guaranteed null terminated) takes priority over RemoteUserIdentifier.

LongRemoteUserIdLength
Length of string pointed to by LongRemoteUserIdPtr - must be set if LongRemoteUserIdPtr is set.

PMQLONG pDataLength
input/output

Pointer to MQLONG. Contains the length of any Security Exit contained in the AgentBuffer upon invocation of the Security Exit. Must be set by a Security Exit to the length of any message being sent in the AgentBuffer or ExitBuffer.

PMQLONG pAgentBufferLength
input

Pointer to MQLONG. The length of the data contained in the AgentBuffer on invocation of the Security Exit.

PMQVOID pAgentBuffer
input/output

On invocation of the Security Exit, this points to any message sent from the partner exit. If ExitResponse2 in the MQCXP structure has the MQXR2_USE_AGENT_BUFFER flag set (default) then a Security Exit needs to set this parameter to point to any message data being sent.

PMQLONG pExitBufferLength
input/output

Pointer to MQLONG. This parameter is initialized to 0 on the first invocation of a Security Exit and the value returned is maintained between calls to the Security Exit during a security exchange.

PMQPTR pExitBufferAddr
input/output

This parameter is initialized to a null pointer on the first invocation of a Security Exit and the value returned is maintained between calls to the Security Exit during a security exchange. If the MQXR2_USE_EXIT_BUFFER flag is set in the ExitResponse2 in the MQCXP structure then a Security Exit needs to set this parameter to point to any message data being sent.

Differences in behavior between security exits defined on CLNTCONN/SVRCONN channel pairs and other channel pairs
Security exits can be defined on all types of channel. However, the behavior of security exits defined on CLNTCONN/SVRCONN channel pairs is slightly different from security exits defined on other channel pairs.

A Security Exit on a CLNTCONN channel can set the Remote User Identifier in the channel definition for processing by a partner SVRCONN exit, or for OAM authorization if no SVRCONN Security Exit is defined and the MCAUSER field of the SVRCONN is not set.

If no CLNTCONN Security Exit is defined then the Remote User Identifier in the channel definition is set to a user identifier from the client environment (which can be blank) by the client MCA.

A security exchange between Security Exits defined on a CLNTCONN and SVRCONN channel pair completes successfully when the SVRCONN Security Exit returns an ExitResponse of MQXCC_OK. A security exchange between other channel pairs completes successfully when the Security Exit which initiated the exchange returns an ExitResponse of MQXCC_OK.

However, the MQXCC_SEND_AND_REQUEST_SEC_MSG ExitResponse code can be used to force continuation of the security exchange: If an ExitResponse of MQXCC_SEND_AND_REQUEST_SEC_MSG is returned by a CLNTCONN or SVRCONN Security Exit then the partner exit must respond by sending a security message (not MQXCC_OK or a null response) or the channel terminates. For Security Exits defined on other types of channel, an ExitResponse of MQXCC_OK returned in response to a MQXCC_SEND_AND_REQUEST_SEC_MSG from the partner Security Exit results in continuation of the security exchange as if a null response was returned and not in termination of the channel.

SSPI security exit
IBM MQ for Windows supplies a security exit that provides authentication for IBM MQ channels by using the Security Services Programming Interface (SSPI). The SSPI provides the integrated security facilities of Windows.

This security exit is for both the IBM MQ client and the IBM MQ server.

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 on Windows, using NTLM authentication services. Two-way authentication is provided on Windows 2000, 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. For more information about using the object or source code of the SSPI security exit, see "Using the SSPI security exit on Windows" on page 1092

Channel send and receive exit programs
You can use the send and receive exits to perform tasks such as data compression and decompression. You can specify a list of send and receive exit programs to be run in succession.

Channel send and receive exit programs are called at the following places in the processing cycle of an MCA:

- The send and receive exit programs are called for initialization at MCA initiation and for termination at MCA termination.
- The send exit program is invoked at one or other end of the channel, depending on the end at which a transmission for one message transfer is sent, immediately before a transmission is sent over the link. Note 4 explains why exits are available in both directions even though message channels send messages in one direction only.
- The receive exit program is invoked at one or other end of the channel, depending on the end at which a transmission for one message transfer is received, immediately after a transmission has been taken from the link. Note 4 explains why exits are available in both directions even though message channels send messages in one direction only.

There might be many transmissions for one message transfer, and there could be many iterations of the send and receive exit programs before a message reaches the message exit at the receiving end.

The channel send and receive exit programs are passed an agent buffer containing the transmission data as sent or received from the communications link. For send exit programs, the first 8 bytes of the buffer are reserved for use by the MCA, and must not be changed. If the program returns a different buffer, then these first 8 bytes must exist in the new buffer. The format of data presented to the exit programs is not defined.

A good response code must be returned by send and receive exit programs. Any other response causes an MCA abnormal end (abend).

Note: Do not issue an MQGET, MQPUT, or MQPUT1 call within sync point from a send or receive exit.

Note:

1. Send and receive exits typically work in pairs. For example a send exit might compress the data and a receive exit decompress it, or a send exit might encrypt the data and a receive exit decrypt it. When you define the appropriate channels, make sure that compatible exit programs are named for both ends of the channel.
2. If compression is turned on for the channel, the exits are passed compressed data.
3. Channel send and receive exits might be called for message segments other than for application data, for example, status messages. They are not called during the startup dialog, nor the security check phase.
4. Although message channels send messages in one direction only, channel-control data, such as heart beats and end of batch processing, flows in both directions, and these exits are available in both

directions, also. However, some of the initial channel startup data flows are exempt from processing by any of the exits.

5. There are circumstances in which send and receive exits could be invoked out of sequence; for example, if you are running a series of exit programs or if you are also running security exits. Then, when the receive exit is first called upon to process data, it might receive data that has not passed through the corresponding send exit. If the receive exit just performed the operation, for example decompression, without first checking that it was required, the results would be unexpected.

   You need to code your send and receive exits in such a way that the receive exit can check that the data it is receiving has been processed by the corresponding send exit. The recommended way to do so is to code your exit programs so that:

   - The send exit sets the value of the ninth byte of data to 0 and shifts all the data along 1 byte, before performing the operation. (The first 8 bytes are reserved for use by the MCA.)
   - If the receive exit receives data that has a 0 in byte 9, it knows that the data has come from the send exit. It removes the 0, performs the complementary operation, and shifts the resulting data back by 1 byte.
   - If the receive exit receives data that has something other than 0 in byte 9, it assumes that the send exit has not run, and sends the data back to the caller unchanged.

   When using security exits, if the channel is ended by the security exit it is possible that a send exit might be called without the corresponding receive exit. One way to prevent this problem is to code the security exit to set a flag, in MQCD.SecurityUserData or MQCD.SendUserData, for example, when the exit decides to end the channel. Then the send exit needs to check this field, and process the data only if the flag is not set. This check prevents the send exit from unnecessarily altering the data, and thus prevents any conversion errors that could occur if the security exit received altered data.

Channel send exit programs - reserving space
You can use send and receive exits to transform the data before transmission. Channel send exit programs can add their own data about the transformation by reserving space in the transmission buffer.

This data is processed by the receive exit program and then removed from the buffer. For example, you might want to encrypt the data and add a security key for decryption.

How you reserve space and use it

When the send exit program is called for initialization, set the *ExitSpace* field of MQXCP to the number of bytes to be reserved. See MQCXP for details. *ExitSpace* can be set only during initialization, that is when *ExitReason* has the value MQXR_INIT. When the send exit is invoked immediately before transmission, with *ExitReason* set to MQXR_XMIT, *ExitSpace* bytes are reserved in the transmission buffer. *ExitSpace* is not supported on z/OS.

The send exit need not use all the reserved space. It can use less than *ExitSpace* bytes or, if the transmission buffer is not full, the exit can use more than the amount reserved. When setting the value of *ExitSpace*, you must leave at least 1 KB for message data in the transmission buffer. Channel performance can be affected if reserved space is used for large amounts of data.

The transmission buffer is normally 32KB long. However, if the channel uses TLS then the transmission buffer size is reduced to 15,352 bytes in order fit within the maximum record length defined by RFC 6101 and the related family of TLS standards. A further 1024 bytes are reserved for use by IBM MQ, so the maximum transmission buffer space usable by send exits is 14,328 bytes.

What happens at the receiving end of the channel

Channel receive exit programs must be set up to be compatible with the corresponding send exits. Receive exits must know the number of bytes in the reserved space and must remove the data in that space.

Multiple send exits

You can specify a list of send and receive exit programs to be run in succession. IBM MQ maintains a total for the space reserved by all the send exits. This total space must leave at least 1 KB for message data in the transmission buffer.

The following example shows how space is allocated for three send exits, called in succession:

1. When called for initialization:
   - Send exit A reserves 1 KB.
   - Send exit B reserves 2 KB.
   - Send exit C reserves 3 KB.
2. The maximum transmission size is 32 KB and the user data is 5 KB long.
3. Exit A is called with 5 KB of data; up to 27 KB are available, because 5 KB is reserved for exits B and C. Exit A adds 1 KB, the amount it reserved.
4. Exit B is called with 6 KB of data; up to 29 KB are available, because 3 KB is reserved for exit C. Exit B adds 1 KB, less than the 2 KB it reserved.
5. Exit C is called with 7 KB of data; up to 32 KB are available. Exit C adds 10K, more than the 3 KB it reserved. This amount is valid, because the total amount of data, 17 KB, is less than the 32 KB maximum.

The maximum transmission buffer size for a channel using TLS is 15,352 bytes, not 32KB. This is because the underlying secure socket transmission segments are limited to 16KB and some of the space is required for TLS record overheads. A further 1024 bytes are reserved for use by IBM MQ, so the maximum transmission buffer space usable by send exits is 14,328 bytes.

Channel message exit programs
You can use the channel message exit to perform tasks such as encryption on the link, validation or substitution of incoming user IDs, message data conversion, journaling, and reference message handling. You can specify a list of message exit programs to be run in succession.

Channel message exit programs are called at the following places in the processing cycle of the MCA:

- At MCA initiation and termination
- Immediately after a sending MCA has issued an MQGET call
- Before the receiving MCA issues an MQPUT call

The message exit is passed an agent buffer containing the transmission queue header MQXQH, and the application message text as retrieved from the queue. The format of MQXQH is given in MQXQH - Transmission-queue header.

Multi If you use reference messages (that is, messages that contain only a header that points to some other object that is to be sent), the message exit recognizes the header, MQRMH. It identifies the object, retrieves it in whatever way is appropriate, appends it to the header, then passes it to the MCA for transmission to the receiving MCA. At the receiving MCA, another message exit recognizes that this message is a reference message, extracts the object, and passes the header on to the destination queue. See “Reference messages and large object transfers” on page 767 and “Running the Reference Message samples” on page 1066 for more information about reference messages and some sample message exits that handle them.

Message exits can return the following responses:

- Send the message (GET exit). The message might have been changed by the exit. (This returns MQXCC_OK.)
- Put the message on the queue (PUT exit). The message might have been changed by the exit. (This returns MQXCC_OK.)

- Do not process the message. The message is placed on the dead-letter queue (undelivered message queue) by the MCA.
- Close the channel.
- Bad return code, which causes the MCA to abnormally end.

Note:

1. Message exits are called once for every complete message transferred, even when the message is split into parts.
2. Linux AIX If you provide a message exit on AIX or Linux, the automatic conversion of user IDs to lowercase characters (described here) does not operate.
3. An exit runs in the same thread as the MCA itself. It also runs inside the same unit of work (UOW) as the MCA because it uses the same connection handle. So any calls made under sync point are committed or backed out by the channel at the end of the batch. For example, one channel message exit program can send notification messages to another and these messages are only committed to the queue when the batch containing the original message is committed.

   Therefore you can issue sync point MQI calls from a channel message exit program.

Message conversion outside the message exit
Before calling the message exit, the receiving MCA performs some conversions on the message. This topic describes the algorithms used to perform the conversions.

Which headers are processed

A conversion routine runs in the MCA of the receiver before the message exit is called. The conversion routine begins with the MQXQH header at the beginning of the message. The conversion routine then processes through the chained headers that follow the MQXQH, performing conversion where necessary. The chained headers can extend beyond the offset contained in the HeaderLength parameter of the MQCXP data that is passed to the message exit of the receiver. The following headers are converted in-place:

- MQXQH (format name " `MQXMIT` ")
- MQMD (this header is part of the MQXQH and has no format name)
- MQMDE (format name " `MQHMDE` ")
- MQDH (format name " `MQHDIST` ")
- MQWIH (format name " `MQHWIH` ")

The following headers are not converted, but are stepped over as the MCA continues to process the chained headers:

- MQDLH (format name " `MQDEAD` ")
- any headers with format names beginning with the three characters 'MQH' (for example " `MQHRF` ") that are not otherwise mentioned

How the headers are processed

The Format parameter of each IBM MQ header is read by the MCA. The Format parameter is 8 bytes within the header, which are 8 single-byte characters containing a name.

The MCA then interprets the data following each header as being of the named type. If the Format is the name of a header type eligible for IBM MQ data conversion, it is converted. If it is another name indicating non-MQ data (for example MQFMT_NONE or MQFMT_STRING) then the MCA stops processing the headers.

What is the MQCXP HeaderLength?

The HeaderLength parameter in the MQCXP data supplied to a message exit is the total length of the MQXQH (which includes the MQMD), MQMDE and MQDH headers at the start of the message. These headers are chained using the 'Format' names and lengths.

MQWIH

Chained headers can extend beyond the HeaderLength into the user data area. The MQWIH header, if it is present, is one of those headers that appear beyond the HeaderLength.

If there is an MQWIH header in the chained headers, it is converted in-place before the message exit of the receiver is called.

Channel message retry exit program
The channel message-retry exit is called when an attempt to open the target queue is unsuccessful. You can use the exit to determine under which circumstances to retry, how many times to retry, and how frequently.

This exit is also called at the receiving end of the channel at MCA initiation and termination.

The channel message-retry exit is passed an agent buffer containing the transmission queue header, MQXQH, and the application message text as retrieved from the queue. The format of MQXQH is given in Overview for MQXQH.

The exit is invoked for all reason codes; the exit determines for which reason codes it wants the MCA to retry, for how many times, and at what intervals. (The value of the message-retry count set when the channel was defined is passed to the exit in the MQCD, but the exit can ignore this value.)

The MsgRetryCount field in MQCXP is incremented by the MCA each time the exit is invoked, and the exit returns either MQXCC_OK with the wait time contained in the MsgRetryInterval field of MQCXP, or MQXCC_SUPPRESS_FUNCTION. Retries continue indefinitely until the exit returns MQXCC_SUPPRESS_FUNCTION in the ExitResponse field of MQCXP. See MQCXP for information about the action taken by the MCA for these completion codes.

If all the retries are unsuccessful, the message is written to the dead-letter queue. If there is no dead-letter queue available, the channel stops.

If you do not define a message-retry exit for a channel and a failure occurs that is likely to be temporary, for example MQRC_Q_FULL, the MCA uses the message-retry count and message-retry intervals set when the channel was defined. If the failure is of a more permanent nature and you have not defined an exit program to handle it, the message is written to the dead-letter queue.

Channel auto-definition exit program
The channel auto-definition exit can be used when a request is received to start a receiver or server-connection channel but no definition for that channel exists (not for IBM MQ for z/OS). It can also be called on all platforms for cluster-sender and cluster-receiver channels to allow definition modification for an instance of the channel.

The channel auto-definition exit can be called on all platforms except z/OS when a request is received to start a receiver or server-connection channel but no channel definition exists. You can use it to modify the supplied default definition for an automatically defined receiver or server-connection channel, SYSTEM.AUTO.RECEIVER, or SYSTEM.AUTO.SVRCON. See Preparing channels for a description of how channel definitions can be created automatically.

The channel auto-definition exit can also be called when a request is received to start a cluster-sender channel. It can be called for cluster-sender and cluster-receiver channels to allow definition modification for this instance of the channel. In this case, the exit also applies to IBM MQ for z/OS. A common use of the channel auto-definition exit is to change the names of message exits (MSGEXIT, RCVEXIT, SCYEXIT, and SENDEXIT) because exit names have different formats on different platforms. If no channel auto-definition exit is specified, the default behavior on z/OS is to examine a distributed exit name of the form *[path]/libraryname(function)* and take up to eight chars of function, if present, or libraryname. On z/OS, a channel auto-definition exit program must alter the fields addressed

by MsgExitPtr, MsgUserDataPtr, SendExitPtr, SendUserDataPtr, ReceiveExitPtr, and ReceiveUserDataPtr, rather than the MsgExit, MsgUserData, SendExit, SendUserData, ReceiveExit and ReceiveUserData fields themselves.

For more information, see Working with auto-defined channels.

As with other channel exits, the parameter list is:

```
MQ_CHANNEL_AUTO_DEF_EXIT (ChannelExitParms, ChannelDefinition)
```

`ChannelExitParms` are described in MQCXP. `ChannelDefinition` is described in MQCD.

MQCD contains the values that are used in the default channel definition if they are not altered by the exit. The exit can modify only a subset of the fields; see MQ_CHANNEL_AUTO_DEF_EXIT. However, attempting to change other fields does not cause an error.

The channel auto-definition exit returns a response of either MQXCC_OK or MQXCC_SUPPRESS_FUNCTION. If neither of these responses is returned, the MCA continues processing as though MQXCC_SUPPRESS_FUNCTION were returned. That is, the auto-definition is abandoned, no new channel definition is created, and the channel cannot start.

ALW *Compiling channel exit programs on AIX, Linux, and Windows systems*

Use the following examples to help you compile channel-exit programs for AIX, Linux, and Windows systems.

Windows

Windows

The compiler and linker command for channel-exit programs on Windows:

```
cl.exe /Ic:\mqm\tools\c\include /nologo /c myexit.c
link.exe /nologo /dll myexit.obj /def:myexit.def /out:myexit.dll
```

AIX and Linux systems

Linux AIX

In these examples `exit` is the library name and `ChannelExit` is the function name. On AIX the export file is called `exit.exp`. These names are used by the channel definition to reference the exit program using the format described in MQCD- channel definition. See also the MSGEXIT parameter of the DEFINE CHANNEL command.

AIX Sample compiler and linker commands for channel exits on AIX:

```
$ xlc_r -q64 -e MQStart -bE:exit.exp -bM:SRE -o /var/mqm/exits64/exit
exit.c -I/usr/mqm/inc
```

Linux Sample compiler and linker commands for channel-exits on Linux where the queue manager is 32 bit:

```
$ gcc -shared -fPIC -o /var/mqm/exits/exit exit.c -I/opt/mqm/inc
```

Linux Sample compiler and linker commands for channel-exits on Linux where the queue manager is 64-bit:

```
$ gcc -m64 -shared -fPIC -o /var/mqm/exits64/exit exit.c -I/opt/mqm/inc
```

On the client, a 32 bit or 64 bit exit can be used. This exit must be linked to mqic_r.

AIX On AIX, all functions that are called by IBM MQ must be exported. A sample export file for this make file:

```
#
!channelExit
MQStart
```

Configuring channel exits

To call the channel exit, you must name it in the channel definition.

Channel exits must be named in the channel definition. You can do this naming when you first define the channels, or you can add the information later using, for example, the MQSC command ALTER CHANNEL. You can also give the channel exit names in the MQCD channel data structure. The format of the exit name depends on your IBM MQ platform; see MQCD or MQSC commands for information.

If the channel definition does not contain a user-exit program name, the user exit is not called.

The channel auto-definition exit is the property of the queue manager, not the individual channel. In order for this exit to be called, it must be named in the queue manager definition. To alter a queue manager definition, use the MQSC command ALTER QMGR.

Writing data-conversion exits

This collection of topics contains information about how to write data-conversion exits.

Note: Not supported in MQSeries for VSE/ESA.

When you do an MQPUT, your application creates the message descriptor (MQMD) of the message. Because IBM 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 topic contains information about the data-conversion exit facility that IBM 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 IBM 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 MQGET.

For a description of the parameters that are passed to the data-conversion exit, and detailed usage notes, see Data conversion 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 IBM MQ data conversion interface (DCI).

For Multicast clients, API exits and data-conversion exits need to be able to run on the client-side because some messages might not go through the queue manager. The following libraries are part of the client packages as well as the server packages:

Table 143. Libraries that are in the client and server packages

Operating system	Libraries
AIX	32 bit & 64 bit: libmqm.a & libmqm_r.a
IBM i	LIBMQM & LIBMQM_R
Linux	32 bit & 64 bit: libmqm.so & libmqm_r.so
Windows	32 bit & 64 bit: mqm.dll & mqm.pdb

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 statements 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 MQ_DATA_CONV_EXIT.

See MQGET 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 like 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 integrity of the queue manager, because it does not run in the queue manager's environment.

Data conversion on z/OS

z/OS

On z/OS, be aware of the following:

- Exit programs can be written in assembly 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 typically 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 sync points, as the results could influence units of work declared by the MCA.
  - Do not update any resources controlled by a resource manager other than IBM MQ for z/OS, including those controlled by CICS Transaction Server.

  For channels with CONVERT=YES, the exit is loaded from the data set referenced by the CSQXLIB DD statement. MQ-supplied exits CSQCBDCI and CSQCBDCO for the IBM MQ CICS Bridge are in SCSQAUTH.

Writing a data-conversion exit program for IBM i

Information about steps to consider when writing MQ data-conversion exit programs for IBM i.

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 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 end 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 949.
   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 "5.a" on page 949. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

c. Near the beginning 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 "5.a" on page 949.

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 IBM MQ job. It is recommended that, for production, data-conversion exit programs be stored in QSYS.

Note:

1. If CVTMQMDTA uses packed structures, all IBM 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 IBM MQ for IBM i ; specify TERASPACE(*YES *TSIFC) in the CRTCMOD and CRTBNDC commands.

Writing a data-conversion exit program for IBM MQ for z/OS

Information about steps to consider when writing data-conversion exit programs for IBM 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 in Valid syntax). If the structures are packed, omit the CMQXCALA macros that are generated. Therefore, consider declaring your structures in such a way that all fields are named and no bytes are skipped; in the example in Valid syntax, 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 IBM MQ stub” on page 990 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 typical way, specifying EXECKEY(CICS) in the definition.

Note: Although the LE/370 runtime libraries are needed for running the CSQUCVX utility (see step “2” on page 950), they are not needed for link-editing or running the data-conversion exit itself (see steps “8” on page 951 and “9” on page 951).

See “Writing IMS bridge applications” on page 71 for information about data conversion within the IBM MQ - IMS bridge.

Linux AIX *Writing a data-conversion exit for IBM MQ for AIX or Linux systems*

Information about steps to consider when writing data-conversion exit programs for IBM MQ for AIX or Linux 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.

   If the data conversion exit is used in a threaded environment, the loadable object must be followed by _r to indicate that it is a threaded version.
2. Create a structure to represent your message. See Valid syntax 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” on page 951. The skeleton source file, and the copy, are read-only.

   The skeleton source file is called amqsvfc0.c.

5. On IBM 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 *MQ_INSTALLATION_PATH*/inc, where *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.. 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 end 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 951.
   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 "7.a" on page 952. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.
   c. Near the beginning 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 951.
8. Compile your exit as a shared library, using MQStart as the entry point. To do this, see "Compiling data-conversion exits on AIX and Linux systems" on page 952.
9. Place the output in the exit directory. The default exit directory is `/var/mqm/exits` for 32 bit systems and `/var/mqm/exits64`, for 64 bit systems. You can change these directories in the qm.ini or mqclient.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 IBM 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.

Linux AIX *Compiling data-conversion exits on AIX and Linux systems*

Examples of how to compile a data conversion exit on AIX and Linux systems.

On all platforms, the entry point to the module is MQStart.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

AIX

AIX

Compile the exit source code by issuing one of the following commands:

32 bit applications

Non-threaded

```
cc -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits/MYFORMAT \
     MYFORMAT.c -I  MQ_INSTALLATION_PATH/inc
```

Threaded

```
xlc_r -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits/MYFORMAT_r \
     MYFORMAT.c -I  MQ_INSTALLATION_PATH/inc
```

64 bit applications

Non-threaded

```
cc -q64 -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits64/MYFORMAT \
     MYFORMAT.c -I MQ_INSTALLATION_PATH/inc
```

Threaded

```
xlc_r -q64 -e MQStart -bE:MYFORMAT.exp -bM:SRE -o /var/mqm/exits64/MYFORMAT_r \
     MYFORMAT.c -I  MQ_INSTALLATION_PATH/inc
```

Linux

Linux

Compile the exit source code by issuing one of the following commands:

31 bit applications

Non-threaded

```
 gcc -m31 -shared -fPIC -o /var/mqm/exits/MYFORMAT MYFORMAT.c \
      -I  MQ_INSTALLATION_PATH/inc
```

Threaded

```
gcc -m31 -shared -fPIC -o /var/mqm/exits/MYFORMAT_r MYFORMAT.c
     -I  MQ_INSTALLATION_PATH/inc
```

32 bit applications

Non-threaded

```
 gcc -m32 -shared -fPIC -o /var/mqm/exits/MYFORMAT MYFORMAT.c
      -I  MQ_INSTALLATION_PATH/inc
```

Threaded

```
gcc -m32 -shared -fPIC -o /var/mqm/exits/MYFORMAT_r MYFORMAT.c
     -I  MQ_INSTALLATION_PATH/inc
```

64 bit applications

Non-threaded

```
gcc -m64 -shared -fPIC -o /var/mqm/exits64/MYFORMAT MYFORMAT.c
     -I  MQ_INSTALLATION_PATH/inc
```

Threaded

```
gcc -m64 -shared -fPIC -o /var/mqm/exits64/MYFORMAT_r MYFORMAT.c
     -I  MQ_INSTALLATION_PATH/inc
```

Windows *Writing a data-conversion exit for IBM MQ for Windows*

Information about steps to consider when writing data-conversion exit programs for IBM 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, *MQ_INSTALLATION_PATH*\Tools\C\Samples. *MQ_INSTALLATION_PATH* is the directory where IBM MQ is installed. 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 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” on page 954.

   amqsvfc0.c is in *MQ_INSTALLATION_PATH*\Tools\C\Samples where *MQ_INSTALLATION_PATH* is the directory where IBM MQ is installed. (The default installation directory is `C:\Program Files\IBM\MQ`.)

   The skeleton includes a sample header file amqsvmha.h in the *MQ_INSTALLATION_PATH*\Tools\C\include 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 end 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 954.

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 "5.a" on page 954. Remove the comment characters to activate the function. If there are several functions, create calls for each of them.

c. Near the beginning 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 954.

6. Create the following command file:

```
cl -I  MQ_INSTALLATION_PATH\Tools\C\Include -Tp   \
MYFORMAT.C


MYFORMAT.DEF
```

where *MQ_INSTALLATION_PATH* is the directory where IBM MQ is installed.

7. Issue the command file to compile your exit as a DLL file.
8. Place the output in the exit subdirectory below the IBM MQ data directory. The default directory for installing your exits on 32 bit systems is *MQ_DATA_PATH*\Exits and for 64 bit systems is *MQ_DATA_PATH*\Exits64

The path used to look for the data-conversion exits is given in the registry. The registry folder is:

```
HKEY_LOCAL_MACHINE\SOFTWARE\IBM\WebSphere
MQ\Installation\MQ_INSTALLATION_NAME\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 IBM 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.

Windows *Exit and switch load files on Windows operating systems*

The IBM WebSphere MQ for Windows 7.5 queue manager processes are 32-bit. As a result, when using 64-bit applications, some types of exit and XA switch load files 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 API call or command fails.

Two attributes are supported in the qm.ini file for *ExitPath*. These are ExitsDefaultPath= *MQ_INSTALLATION_PATH*\exits and ExitsDefaultPath64= *MQ_INSTALLATION_PATH*\exits64. *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed. Using these ensures that the appropriate library can be found. If an exit is used in an IBM MQ cluster, this also ensures that the appropriate library on a remote system can 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 exit	32-bit and 64-bit	64-bit
Data conversion exit	32-bit	64-bit
Server Channel exits (all types)	64-bit	64-bit
Client Channel exits (all types)	32-bit	64-bit
Installable service exit	64-bit	64-bit
Cluster WLM exit	64-bit	64-bit
Pub/Sub routing exit	64-bit	64-bit
Database switch load files	32-bit and 64-bit	64-bit
External Transaction Manager AX libraries	32-bit	64-bit
Pre-connect exit	32-bit	64-bit

Referencing connection definitions using a pre-connect exit from a repository

IBM MQ MQI clients can be configured to look up a repository to obtain connection definitions using a pre-connect exit library.

Introduction

A client application can connect to a queue manager using client channel definition tables (CCDT). Generally, the CCDT file is located on a central network file server, and have clients referencing it. Since it is difficult to manage and administer various client applications referencing the CCDT file, a flexible approach is to store the client definitions in a global repository like an LDAP directory, a WebSphere Registry and Repository or any other repository. Storing the client connection definitions in a repository makes managing client connection definitions easier, and applications can access the correct and most current client connection definitions.

During the MQCONN/X call execution, the IBM MQ MQI client loads an application specified pre-connect exit library, and invokes an exit function to retrieve connection definitions. The retrieved connection definitions are then used to establish connection to a queue manager. The details of exit library and function to invoke are specified in the mqclient.ini configuration file.

Syntax

`void MQ_PRECONNECT_EXIT` (*pExitParms*, *pQMgrName*, *ppConnectOpts*, *pCompCode*, *pReason*);

Parameters

pExitParms

Type: PMQNXP input /output

The **PreConnection** exit parameter structure.

The structure is allocated and maintained by the caller of the exit.

pQMgrName

Type: PMQCHAR input/output

Name of the queue manager.

On input, this parameter is the filter string supplied to the MQCONN API call through the **QMgrName** parameter. This field might be blank, explicit, or contain certain wildcard characters. The field is changed by the exit. The parameter is NULL when the exit is called with MQXR_TERM.

ppConnectOpts
Type: ppConnectOpts input/output

Options that control the action of MQCONNX.

This is a pointer to an MQCNO connection options structure that controls the action of the MQCONN API call. The parameter is NULL when the exit is called with MQXR_TERM. The MQI client always provides an MQCNO structure to the exit, even if it was not originally provided by the application. If an application provides an MQCNO structure, the client makes a duplicate to pass it to the exit where it is modified. The client retains the ownership of the MQCNO.

An MQCD referenced through the MQCNO takes precedence over any connection definition provided through the array. The client uses the MQCNO structure to connect to the queue manager and the others are ignored.

pCompCode
Type: PMQLONG input/output

Completion code.

Pointer to an MQLONG that receives the exits completion code. It must be one of the following values:

- `MQCC_OK` - Successful completion
- `MQCC_WARNING` - Warning (partial completion)
- `MQCC_FAILED` - Call failed

pReason
Type: PMQLONG input/output

Reason qualifying pCompCode.

Pointer to an MQLONG that receives the exit reason 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 Invocation

```
void MQ_PRECONNECT_EXIT (&ExitParms, &QMgrName, &pConnectOpts, &CompCode, &Reason);
```

`Parameter`

```
PMQNXP  pExitParms   /*PreConnect exit parameter structure*/
PMQCHAR pQMgrName    /*Name of the queue manager*/
PPMQCNO ppConnectOpts/*Options controlling the action of MQCONNX*/
PMQLONG pCompCode    /*Completion code*/
PMQLONG pReason      /*Reason qualifying pCompCode*/
```

Writing and compiling publish exits

You can configure a publish exit at the queue manager to change the contents of a published message before it is received by subscribers. You can also change the message header, or not deliver the message to a subscription.

Note: Publish exits are not supported on z/OS.

You can use the publish exit to inspect and alter messages delivered to subscribers:

- Examine the contents of a message published to each subscriber
- Modify the contents of a message published to each subscriber
- Alter the queue to which a message is put

- Stop the delivery of a message to a subscriber

Writing a publish exit

Use the steps in "Writing exits and installable services on AIX, Linux, and Windows" on page 902, to help you write and compile your exit.

The provider of the publish exit defines what the exit does. The exit, however, must conform to the rules defined in MQPSXP.

IBM MQ does not provide an implementation of the MQ_PUBLISH_EXIT entry point. It does provide a C language typedef declaration. Use the typedef to declare the parameters to a user-written exit correctly. The following example illustrates how to use the typedef declaration:

```
#include "cmqec.h"

MQ_PUBLISH_EXIT MyPublishExit;

void MQENTRY MyPublishExit( PMQPSXP pExitParms,
                            PMQPBC  pPubContext,
                            PMQSBC  pSubContext )
{
/* C language statements to perform the function of the exit */
}
```

The publish exit runs within the queue manager process, as a result of the following operations:

- A Publish operation where a message is delivered to one or more subscribers
- A Subscribe operation where one or more retained messages are delivered
- A Subscription Request operation where one or more retained messages are delivered

If the publish exit is called for a connection, the first time that it is called an *ExitReason* code of MQXR_INIT is set. Before the connection disconnects after using a publish exit, the exit is called with an *ExitReason* code of MQXR_TERM.

If the publish exit is configured, but cannot be loaded when the queue manager is started, publish/subscribe message operations are inhibited for the queue manager. You must fix the problem or restart the queue manager before publish/subscribe messaging is re-enabled.

Each IBM MQ connection that requires the publish exit might fail to load or initialize the exit. If the exit fails to load or initialize, publish/subscribe operations that require the publish exit are disabled for that connection. The operations fail with the IBM MQ reason code MQRC_PUBLISH_EXIT_ERROR.

The context in which the publish exit is called is the connection by an application to the queue manager. A user data area is maintained by queue manager for each connection that is performing publish operations. The exit can retain information in the user data area for each connection.

A publish exit can use some MQI calls. It can only use those MQI calls that manipulate message properties. The calls are:

- MQBUFMH
- MQCRTMH
- MQDLTMH
- MQDLTMP
- MQMHBUF
- MQINQMP
- MQSETMP

If the publish exit changes the destination queue manager or queue name, no new authority check is carried out.

Compiling a publish exit

The publish exit is a dynamically loaded library; it can be thought of as a channel-exit. For information about compiling exits, see "Writing exits and installable services on AIX, Linux, and Windows" on page 902.

Sample publish exit

The sample exit program is called `amqspse0.c`. It writes a different message to a log file depending on whether the exit was called for initialize, publish, or terminate operations. It also demonstrates the use of the exit user area field to allocate and free storage appropriately.

Configuring publish exits

You must define certain attributes to configure a publish exit.

On Windows and Linux you can use the IBM MQ explorer to define the attributes. The attributes are defined on the queue manager properties page, under Publish/Subscribe.

To configure the publish exit in the `qm.ini` file on AIX and Linux systems, create a stanza called `PublishSubscribe`. The `PublishSubscribe` stanza has the following attributes:

PublishExitPath=[path]|*module_name*
Module name and path containing the publish exit code. The maximum length of this field is `MQ_EXIT_NAME_LENGTH`. The default is no publish exit.

PublishExitFunction= *function_name*
Name of the function entry point into the module that contains the publish exit code. The maximum length of this field is `MQ_EXIT_NAME_LENGTH`.

IBM i On IBM i, if a program is used, omit `PublishExitFunction`.

PublishExitData= *string*
If the queue manager is calling a publish exit, it passes an `MQPSXP` structure as input. The data specified using the **PublishExitData** attribute is provided in the *ExitData* field of the structure. The string can be up to `MQ_EXIT_DATA_LENGTH` characters in length. The default is 32 blank characters.

Writing and compiling cluster workload exits

Write a cluster workload exit program to customize the workload management of clusters. You might take the cost of using a channel at different times of day, or message content, into account when routing messages. These are factors that are not considered by the standard workload management algorithm.

In most cases the workload management algorithm is sufficient for your needs. However, so that you can provide your own user-exit program to tailor workload management, IBM MQ includes a user exit, the cluster workload exit.

You might have some particular information about your network or messages that you could use to influence workload balancing. You might know which are the high-capacity channels or the cheap network routes, or you might want to route messages depending upon their content. You could decide to write a cluster workload exit program, or use one supplied by a third party.

The cluster workload exit is called when accessing a cluster queue. It is called by `MQOPEN`, `MQPUT1` and `MQPUT`.

The target queue manager selected at `MQOPEN` time is fixed if `MQOO_BIND_ON_OPEN` is specified. In this case the exit is run only once.

If the target queue manager is not fixed at `MQOPEN` time, the target queue manager is chosen at the time of the `MQPUT` call. If the target queue manager is not available, or fails while the message is still on the transmission queue, the exit is called again. A new target queue manager is selected. If the message channel fails while the message is being transferred, and the message is backed out, a new target queue manager is selected.

Multi On Multiplatforms, the queue manager loads the new cluster workload exit the next time the queue manager is started.

If the queue manager definition does not contain a cluster workload exit program name, the cluster workload exit is not called.

Various data are passed to a cluster workload exit in the exit parameter structure, MQWXP:

- The message definition structure, MQMD.
- The message length parameter.
- A copy of the message, or part of the message.

On non-z/OS platforms, if you use CLWLMode=FAST, each operating system process loads its own copy of the exit. Different connections to the queue manager can cause different copies of the exit to be invoked. If the exit is run in the default safe mode, CLWLMode=SAFE, a single copy of the exit runs in its own separate process.

Writing cluster workload exits

z/OS For information about writing cluster workload exits for z/OS, see "Cluster workload exit programming for IBM MQ for z/OS" on page 962.

From IBM MQ 9.1.0, cluster workload exits run in the channel initiator address space, instead of the queue manager address space. If you have a cluster workload exit, you should remove the CSQXLIB DD statement from your queue manager started task procedure, and add the dataset containing the cluster workload exit to the CSQXLIB concatenation on your channel initiator started task procedure.

Multi For Multiplatforms, cluster workload exits must not use MQI calls. In other respects, the rules for writing and compiling cluster workload exit programs are like the rules that apply to channel exit programs. Follow the steps in "Writing exits and installable services on AIX, Linux, and Windows" on page 902, and use the sample program, "Sample cluster workload exit" on page 960 to help write and compile your exit.

For more information about channel exits, see "Writing channel-exit programs" on page 929.

Configuring cluster workload exits

You name cluster workload exits in the queue manager definition by specifying the cluster workload exit attribute on the ALTER QMGR command. For example:

```
ALTER QMGR CLWLEXIT(myexit)
```

Related reference

Cluster workload exit call and data structures

Sample cluster workload exit

IBM MQ includes a sample cluster workload exit program. You can copy the sample and use it as a basis for your own programs.

z/OS **IBM MQ for z/OS**

The sample cluster workload exit program is supplied in Assembler and in C. The Assembler version is called CSQ4BAF1 and can be found in the library `thlqual.SCSQASMS`. The C version is called CSQ4BCF1 and can be found in the library `thlqual.SCSQC37S`. `thlqual` is the target library high-level qualifier for IBM MQ data sets in your installation.

Multi **IBM MQ for Multiplatforms**

The sample cluster workload exit program is supplied in C and is called `amqswlm0.c`. It can be found in:

Table 144. Sample cluster workload exit program location for Multiplatforms	
Platform	**Filepath**
AIX AIX	*MQ_INSTALLATION_PATH*/samp
Windows Windows	*MQ_INSTALLATION_PATH* \Tools\c\Samples
IBM i IBM i	The qmqm library

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

This sample exit routes all messages to a particular queue manager, unless that queue manager becomes unavailable. It reacts to the failure of the queue manager by routing messages to another queue manager.

Indicate which queue manager you want messages to be sent to. Supply the name of the cluster-receiver channel in the CLWLDATA attribute on the queue manager definition. For example:

```
ALTER QMGR CLWLDATA(' my-cluster-name. my-queue-manager ')
```

To enable the exit, supply its full path and name in the CLWLEXIT attribute:

Linux AIX On AIX and Linux:

```
ALTER QMGR CLWLEXIT(' path /amqswlm(clwlFunction)')
```

Windows On Windows:

```
ALTER QMGR CLWLEXIT(' path \amqswlm(clwlFunction)')
```

z/OS On z/OS:

```
ALTER QMGR CLWLEXIT(CSQ4BxF1)
```

where x is either 'A' or 'C', depending on the programming language of the version you are using.

IBM i On IBM i, use either of the following commands:

- Use the MQSC command:

  ```
  ALTER QMGR CLWLEXIT('AMQSWLM       library       ')
  ```

  Both the program name and the library name occupy 10 characters and are blank-padded to the right if necessary.
- Use the CL command:

  ```
  CHGMQM MQMNAME( qmgrname ) CLWLEXIT(' library /AMQSWLM')
  ```

Now, instead of using the supplied workload management algorithm, IBM MQ calls this exit to route all messages to your chosen queue manager.

z/OS *Cluster workload exit programming for IBM MQ for z/OS*

Cluster workload exits are invoked as if by a z/OS **LINK** command. Exits are subject to a number of stringent programming rules. Avoid using most SVC commands that involve waits, or using a STAE or ESTAE in a workload exit.

Cluster workload exits are invoked as if by a 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
- Storage key 8
- Program Key Mask 8
- TCB key 8

Put the link-edited modules in the data set specified by the `CSQXLIB DD` statement of the started task procedure of the channel initiator. The names of the load modules are specified as the workload exit names in the queue manager definition.

When writing workload exits for IBM MQ for z/OS, the following rules apply:

- You must write exits in assembler or C. If you use C, it must conform to the C systems programming environment for system exits, described in the *z/OS C/C++ Programming Guide*, SC09-4765.
- If using the `MQXCLWLN` call, link edit with `CSQMFCLW`, supplied in *`thlqual`*`.SCSQLOAD`.
- Exits are loaded from the non-authorized libraries defined by a `CSQXLIB DD` statement. Providing `CSQXLIB` has `DISP=SHR`, exits can be updated while the queue manager is running, with the new version used in the next `MQCONN` thread the queue manager starts.
- Exits must be reentrant, and capable of running anywhere in virtual storage.
- Exits must reset the environment on return to that at entry.
- Exits must free any storage obtained, or ensure that storage is freed by a subsequent exit invocation.
- No MQI calls are allowed.
- Exits must not use any system services that could cause a wait, because a wait severely degrades the performance of the queue manager. In general, therefore, avoid an SVC, PC, or I/O.
- Exits must not issue an ESTAE or SPIE, apart from within any subtasks they attach.

Note: There are no absolute restrictions on what you can do in an exit. However, most SVCs involve waits, so avoid them, except for the following commands:

- **GETMAIN** / **FREEMAIN**
- **LOAD** / **DELETE**

Do not use ESTAEs and ESPIEs because their error handling might interfere with the error handling performed by IBM MQ. IBM MQ might not be able to recover from an error, or your exit program might not receive all the error information.

The system parameter `EXITLIM` limits the amount of time an exit might run for. The default value for `EXITLIM` is 30 seconds. If you see the return code `MQRC_CLUSTER_EXIT_ERROR`, 2266 `X'8DA'` your exit might be looping. If you think the exit needs more than 30 seconds to complete, increase the value of `EXITLIM`.

Building a procedural application

You can write an IBM MQ application in one of several procedural languages, and run the application on several different platforms.

AIX Building your procedural application on AIX

The AIX publications describe how to build executable applications from the programs that you write.

This topic describes the additional tasks, and the changes to the standard tasks, that you must perform when building IBM MQ for AIX applications to run under AIX. C, C++, and COBOL are supported. For information about preparing your C++ programs, see Using C++.

The tasks that you must perform to create an executable application using IBM 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 IBM MQ for AIX include files for the language that you are using. Make yourself familiar with the contents of these files. See “IBM MQ data definition files” on page 693 for a full description.

When you run threaded server or threaded client applications, set the environment variable AIXTHREAD_SCOPE=S.

AIX *Preparing C programs in AIX*

Prepare and compile C programs in AIX using the ANSI compiler, the XLC 17 compiler, or the VisualAge C/C++ compiler.

Before you begin

In the following code examples, *${MQ_INSTALLATION_PATH}* represents the high-level directory in which IBM MQ is installed. You can either use the code unchanged, to pull in the value that you have set for variable *MQ_INSTALLATION_PATH*, or you can replace *${MQ_INSTALLATION_PATH}* with the actual path.

You need to link your program with one of the following library files:

Library file	Program/exit type
libmqm.a	Server for C (unthreaded application)
libmqm_r.a	Server for C (threaded application)
libmqic.a & libmqm.a	Client for C (unthreaded application)
libmqic_r.a & libmqm_r.a	Client for C (threaded application)

Note:

1. You cannot link to more than one library. That is, you cannot link to both a threaded and a non-threaded library at the same time.
2. If you are writing an installable service (see the Administering IBM MQ 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.
3. 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.
4. You need to link trusted applications to the threaded IBM MQ libraries. However, only one thread in a trusted application on IBM MQ for AIX or Linux systems can be connected at a time.
5. You must link IBM MQ libraries before any other product libraries.

About this task

Sample C programs are supplied in the *${MQ_INSTALLATION_PATH}*/`samp/bin` directory.

Each of the following code examples uses the `amqsput0` sample program with the ANSI compiler or the XLC 17 compiler, and links with one of the Server for C library files. If you want to use the program on a machine that has only the IBM MQ MQI client for AIX installed, alter the example code to link with one of the client library (`-lmqic`) files instead.

If you are using the VisualAge C/C++ compiler for C++ programs, include the option `-q namemangling=v5` to get all the IBM MQ symbols resolved when linking the libraries.

For more information about writing 64-bit applications, see Coding standards on 64-bit platforms.

Example

To compile a sample C program from a single compilation unit, run the appropriate command from the following list:

32 bit non-threaded application

```
xlc -o amqsput_32 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib
-lmqm
```

V 9.4.0 **32 bit non-threaded application using XLC 17 compiler**

```
ibm-clang -o amqsput_32 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib -lmqm
```

32 bit threaded application

```
xlc_r -o amqsputc_32_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib -lmqm_r
```

V 9.4.0 **32 bit threaded application using XLC 17 compiler**

```
ibm-clang_r -o amqsput_32_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib -lmqm_r
```

64 bit non-threaded application

```
xlc -q64 -o amqsput_64 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/
lib64 -lmqm
```

V 9.4.0 **64 bit non-threaded application using XLC 17 compiler**

```
ibm-clang -m64 -o amqsput_64 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib64 -lmqm
```

64 bit threaded application

```
xlc_r -q64 -o amqsputc_64_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib64 -lmqm_r
```

V 9.4.0 **64 bit threaded application using XLC 17 compiler**

```
ibm-clang_r -m64 -o amqsput_64_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib64 -lmqm_r
```

AIX

Preparing COBOL programs in AIX

Prepare and compile COBOL programs in AIX using IBM COBOL Set or Micro Focus COBOL.

Before you begin

In the following code examples, *${MQ_INSTALLATION_PATH}* represents the high-level directory in which IBM MQ is installed. You can either use the code unchanged, to pull in the value that you have set

for variable *MQ_INSTALLATION_PATH*, or you can replace *${MQ_INSTALLATION_PATH}* with the actual path.

- 32 bit COBOL copy books are installed in directory *${MQ_INSTALLATION_PATH}*/inc/cobcpy32, and symbolic links are created in directory *${MQ_INSTALLATION_PATH}*/inc
- 64 bit COBOL copy books are installed in directory *${MQ_INSTALLATION_PATH}*/inc/cobcpy64

You need to link your program with one of the following library files:

Library file	Program/exit type
libmqmcb.a	Server for COBOL (unthreaded application)
libmqmcb_r.a	Server for COBOL (threaded application)
libmqicb.a	Client for COBOL (unthreaded application
libmqicb_r.a	Client for COBOL (threaded application)

About this task

Sample COBOL programs are supplied with IBM MQ, and the following code examples use these programs. You can use the IBM COBOL Set compiler or Micro Focus COBOL compiler, depending on the program:

- Programs beginning amq0 are suitable for either compiler.
- Programs beginning amqm are suitable for the Micro Focus COBOL compiler.

Example

Preparing COBOL programs using IBM COBOL Set for AIX

To compile a sample COBOL program, run the appropriate command from the following list:

32 bit non-threaded server application

```
cob2 -o amq0put0 amq0put0.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqmcb -qLIB \
-I${MQ_INSTALLATION_PATH}/inc/cobcpy32
```

32 bit non-threaded client application

```
cob2 -o amq0put0 amq0put0.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqicb -qLIB \
-I${MQ_INSTALLATION_PATH}/inc/cobcpy32
```

32 bit threaded server application

```
cob2_r -o amq0put0 amq0put0.cbl -qTHREAD -L${MQ_INSTALLATION_PATH}/lib \
-lmqmcb_r -qLIB -I${MQ_INSTALLATION_PATH}/inc/cobcpy32
```

32 bit threaded client application

```
cob2_r -o amq0put0 amq0put0.cbl -qTHREAD -L${MQ_INSTALLATION_PATH}/lib \
-lmqicb_r -qLIB -I${MQ_INSTALLATION_PATH}/inc/cobcpy32
```

64 bit non-threaded server application

```
cob2 -o amq0put0 amq0put0.cbl -q64 -L${MQ_INSTALLATION_PATH}/lib -lmqmcb \
-qLIB -I${MQ_INSTALLATION_PATH}/inc/cobcpy64
```

64 bit non-threaded client application

```
cob2 -o amq0put0 amq0put0.cbl -q64 -L${MQ_INSTALLATION_PATH}/lib -lmqicb \
-qLIB -I${MQ_INSTALLATION_PATH}/inc/cobcpy64
```

64 bit threaded server application

```
cob2_r -o amq0put0 amq0put0.cbl -q64 -qTHREAD -L${MQ_INSTALLATION_PATH}/lib \
-lmqmcb_r -qLIB -I${MQ_INSTALLATION_PATH}/inc/cobcpy64
```

64 bit threaded client application

```
cob2_r -o amq0put0 amq0put0.cbl -q64 -qTHREAD -L${MQ_INSTALLATION_PATH}/lib \
-lmqicb_r -qLIB -I${MQ_INSTALLATION_PATH}/inc/cobcpy64
```

Preparing COBOL programs using Micro Focus COBOL

For a description of the environment variables that you need to set up, see the Micro Focus COBOL documentation.

To compile a 32 bit COBOL program using Micro Focus COBOL, first set the COBCPY environment variable:

```
export COBCPY=${MQ_INSTALLATION_PATH}/inc/cobcpy32
```

then enter the appropriate command from the following list:

32 bit non-threaded server application

```
cob32 -xvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqmcb
```

32 bit non-threaded client application

```
cob32 -xvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqicb
```

32 bit threaded server application

```
cob32 -xtvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqmcb_r
```

32 bit threaded client application

```
cob32 -xtvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqicb_r
```

To compile a 64 bit COBOL program using Micro Focus COBOL, first set the COBCPY environment variable:

```
export COBCPY=${MQ_INSTALLATION_PATH}/inc/cobcpy64
```

then enter the appropriate command from the following list:

64 bit non-threaded server application

```
cob64 -xvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib64 -lmqmcb
```

64 bit non-threaded client application

```
cob64 -xvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib64 -lmqicb
```

64 bit threaded server application

```
cob64 -xtvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib64 -lmqmcb_r
```

64 bit threaded client application

```
cob64 -xtvP amqminqx.cbl -L${MQ_INSTALLATION_PATH}/lib64 -lmqicb_r
```

AIX *Preparing CICS application programs in AIX*

Use this information when preparing CICS programs in AIX.

Use *XA switch* modules to link CICS with IBM MQ. For more information on the XA switch structure, see The XA switch structures.

The sample source code file is provided to enable you to develop the XA switches for other transaction messages. The name of the switch load module provided is listed inTable 145 on page 967.

Table 145. Essential code for CICS application programs on AIX: XA initialization routine

Description	C (source)	C (exec) - add to your XAD.Stanza
XA initialization routine	amqzscix.c	*amqzsc* - CICS for AIX

Use the prebuilt version of the IBM MQ switch load file *amqzsc*, which is provided with the product.

Always link your C transactions with the threadsafe IBM 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 Administering IBM MQ IBM MQ System Administration Guide.

AIX *TXSeries CICS support*

IBM MQ on AIX supports TXSeries CICS using the XA interface. Ensure that CICS applications are linked to the threaded version of the IBM MQ libraries.

You can run CICS programs using IBM COBOL Set for AIX or Micro Focus COBOL. The following sections describe the difference between running CICS programs on IBM COBOL Set for AIX and Micro Focus COBOL.

Write IBM MQ programs that are loaded into the same CICS region in either C or COBOL. You cannot make a combination of C and COBOL MQI calls into the same CICS region. Most MQI calls in the second language used fail with a reason code of `MQRC_HOBJ_ERROR`.

Preparing CICS COBOL programs using IBM COBOL Set for AIX

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

To use IBM COBOL, follow these steps:

1. Export the following environment variable:

```
export LDFLAGS="-qLIB -bI:/usr/lpp/cics/lib/cicsprIBMCOB.exp \
                -I${MQ_INSTALLATION_PATH}/inc -I/usr/lpp/cics/include \
                -e _iwz_cobol_main  \
```

   where `LIB` is a compiler directive.

2. Translate, compile, and link the program:

```
cicstcl -l IBMCOB yourprog.ccp
```

Preparing CICS COBOL programs using Micro Focus COBOL

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

To use Micro Focus COBOL, follow these steps:

1. Add the IBM MQ COBOL runtime library module to the runtime library:

```
cicsmkcobol -L/usr/lib/dce -L${MQ_INSTALLATION_PATH}/lib \
             ${MQ_INSTALLATION_PATH}/lib/libmqmcbrt.o -lmqe_r
```

   Note: With `cicsmkcobol`, IBM 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 recommended to move these functions from the COBOL applications to your own library, for example, `myMQ.so`. After moving the functions, do not include the IBM 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 runtime COBOL library to call IBM MQ for AIX or Linux systems.

Note: Run `cicsmkcobol` only when you install one of the following products:

- 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 IBM MQ

2. Export the following environment variable:

```
COBCPY=${MQ_INSTALLATION_PATH}/inc export COBCPY
```

3. Translate, compile, and link the program:

```
cicstcl -l COBOL -e yourprog.ccp
```

Preparing CICS C programs

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Build CICS C programs using the standard CICS facilities:

1. Export ***one*** of the following environment variables:

   - `LDFLAGS="-L${MQ_INSTALLATION_PATH}/lib -lmqm_r" export LDFLAGS`
   - `USERLIB="-L${MQ_INSTALLATION_PATH}/lib -lmqm_r" export USERLIB`

2. Translate, compile, and link the program:

```
cicstcl -l C amqscic0.ccs
```

CICS C sample transaction

Sample C source for a AIX IBM 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 with a queue name that 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.

IBM i Building your procedural application on IBM i

The IBM i publications describe how to build executable applications from the programs that you write, to run with IBM i on iSeries or System i systems.

This topic describes the additional tasks, and the changes to the standard tasks, that you must perform when building IBM MQ for IBM i procedural applications to run on IBM i systems. COBOL, C, C++, Java and RPG programming languages are supported. For information about preparing your C++ programs, see Using C++. For information about preparing your Java programs, see Using IBM MQ classes for Java.

The tasks that you must perform to create an executable IBM MQ for IBM i 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 IBM MQ for IBM i data definition files for the language that you are using. Make yourself familiar with the contents of these files. See “IBM MQ data definition files” on page 693 for a full description.

IBM i *Preparing C programs in IBM i*

IBM MQ for IBM i supports messages up to 100 MB in size. Application programs written in ILE C, supporting IBM MQ messages greater than 16 MB, need to use the `Teraspace` compiler option to allocate sufficient memory for these messages.

For more information about the C compiler options, see the *WebSphere Development Studio ILE C/C++ Programmer's Guide*.

To compile a C module, you can use the IBM i command **CRTCMOD**. Make sure that the library containing the include files (QMQM) is in the library list when you compile.

You must then bind the output of the compiler with the service program using the **CRTPGM** command.

Table 146. Example of CRTPGM in non-threaded and threaded environments

Type of environment	Command	Program/exit type

Non-threaded environment	`CRTPGM PGM( pgmname )`
	`MODULE( pgmname )`
	`BNDSRVPGM(QMQM/LIBMQM)`
		Server or client for C

Threaded environment	`CRTPGM PGM( pgmname )`
	`MODULE( pgmname )`
	`BNDSRVPGM(QMQM/LIBMQM_R)`
		Server or client for C

where *pgmname* is the name of your program.

Table 147 on page 969 lists the libraries that are needed when preparing C programs on IBM i in a non-threaded environment and threaded environment.

Table 147. Libraries needed for non-threaded and threaded environments

Type of environment	Library file	Program/exit type
Non-threaded environment	LIBMQM	Server for C
	LIBMQIC & LIBMQM	Client for C
Threaded environment	LIBMQM_R	Server for C
	LIBMQIC_R & LIBMQM_R	Client for C

IBM i *Preparing COBOL programs in IBM i*

Learn about preparing COBOL programs in IBM i and the method of accessing the MQI from within the COBOL program.

About this task

For accessing the MQI from within COBOL programs, IBM MQ for IBM i provides a bound procedural call interface provided by service programs. This provides access to all the MQI functions in IBM MQ for IBM i and support for threaded applications. This interface can be used only with the ILE COBOL compiler.

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 IBM i COBOL compilers assume that the delimiter is the quotation mark ("). 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.

Note: The dynamic call interface is not provided in IBM MQ 9.0 or later.

To use the bound procedure call interface, complete the following steps.

Procedure

1. Create a module using the **CRTCBLMOD** compiler specifying the parameter:

   ```
   LINKLIT(*PRC)
   ```

2. Use the **CRTPGM** command to create the program object, specifying the appropriate parameter:

   For non-threaded applications:

   ```
   BNDSRVPGM(QMQM/AMQ0STUB)        Server for COBOL for non-threaded applications
   BNDSRVPGM(QMQM/AMQCSTUB)        Client for COBOL for non-threaded applications
   ```

   For threaded applications:

   ```
   BNDSRVPGM(QMQM/AMQ0STUB_R)        Server for COBOL for threaded applications
   BNDSRVPGM(QMQM/AMQCSTUB_R)        Client for COBOL for threaded applications
   ```

   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 IBM 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 affect overall performance.

   See the *WebSphere Development Studio: ILE COBOL Programmer's Guide* and the *WebSphere Development Studio: ILE COBOL Reference* for further information.

   For more information about compiling a CICS application, see the *CICS for IBM i Application Programming Guide*, SC41-5454.

IBM i *Preparing CICS programs in IBM i*

Learn about the steps required when preparing CICS programs in IBM i.

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 IBM 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 IBM 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)
```

IBM i *Preparing RPG programs in IBM i*

If you are using IBM MQ for IBM i, you can write your applications in RPG.

For more information see "Coding IBM MQ programs in RPG (IBM i only)" on page 1017, and refer to the IBM i Application Programming Reference (ILE/RPG).

IBM i *SQL programming considerations for IBM i*

Learn about the steps required when building an application on IBM i using SQL.

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 IBM 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 IBM 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)
```

Linux Building your procedural application on Linux

This information describes the additional tasks, and the changes to the standard tasks, that you must perform when building IBM MQ for Linux applications to run.

C and C++ are supported. For information about preparing your C++ programs, see Using C++.

Linux *Preparing C programs in Linux*

Precompiled C programs are supplied in the *MQ_INSTALLATION_PATH*`/samp/bin` directory. To build a sample from source code, use the `gcc` compiler.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Work in your normal environment. For further information about programming 64 bit applications, see Coding standards on 64-bit platforms.

Linking libraries

The following tables lists the libraries that are needed when preparing C programs on Linux.

- You need to link your programs with the appropriate library provided by IBM MQ.

  In a non-threaded environment, link to only one of the following libraries:

Library file	Program/exit type
libmqm.so	Server for C
libmqic.so & libmqm.so	Client for C

In a threaded environment, link to only one of the following libraries:

Library file	Program/exit type
libmqm_r.so	Server for C
libmqic_r.so & libmqm_r.so	Client for C

Note:

1. You cannot link to more than one library. That is, you cannot link to both a threaded and a non-threaded library at the same time.
2. If you are writing an installable service (see the Administering IBM MQ for further information), you need to link to the `libmqmzf.so` library.
3. 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.
4. You must link IBM MQ libraries before any other product libraries.

Linux *Building 31-bit applications*

This topic contains examples of the commands used to build 31-bit programs in various environments.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

C client application, 31-bit, non-threaded

```
gcc -m31 -o famqsputc_32 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -LMQ_INSTALLATION_PATH/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=/usr/lib -lmqic
```

C client application, 31-bit, threaded

```
gcc -m31 -o amqsputc_32_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc
-LMQ_INSTALLATION_PATH/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=/usr/lib -lmqic_r -lpthread
```

C server application, 31-bit, non-threaded

```
gcc -m31 -o amqsput_32 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=/usr/lib -lmqm
```

C server application, 31-bit, threaded

```
gcc -m31 -o amqsput_32_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=$
{MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=$
{MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=$
{MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=/usr/lib -lmqm_r -lpthread
```

Linux *Building 32-bit applications*

This topic contains examples of the commands used to build 32-bit programs in various environments.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

C client application, 32-bit, non-threaded

```
gcc -m32 -o amqsputc_32 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=/usr/lib -lmqic
```

C client application, 32-bit, threaded

```
gcc -m32 -o amqsputc_32_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=/usr/lib -lmqic_r -lpthread
```

C server application, 32-bit, non-threaded

```
gcc -m32 -o amqsput_32 amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=/usr/lib -lmqm
```

C server application, 32-bit, threaded

```
gcc -m32 -o amqsput_32_r amqsput0.c -I${MQ_INSTALLATION_PATH}/inc -L$
{MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=$
{MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc -L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=$
{MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/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$
{MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib -Wl,-rpath=${MQ_INSTALLATION_PATH}/lib
-Wl,-rpath=/usr/lib -lmqm_r -lpthread
```

Linux *Building 64-bit applications*

This topic contains examples of the commands used to build 64-bit programs in various environments.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

C client application, 64-bit, non-threaded

```
gcc -m64 -o amqsputc_64 amqsput0.c
-I${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=/usr/lib64 -lmqic
```

C client application, 64-bit, threaded

```
gcc -m64 -o amqsputc_64_r amqsput0.c
-I${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=/usr/lib64 -lmqic_r -lpthread
```

C server application, 64-bit, non-threaded

```
gcc -m64 -o amqsput_64 amqsput0.c
-I${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=/usr/lib64 -lmqm
```

C server application, 64-bit, threaded

```
gcc -m64 -o amqsput_64_r amqsput0.c
-I${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/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${MQ_INSTALLATION_PATH}/inc
-L${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=${MQ_INSTALLATION_PATH}/lib64
-Wl,-rpath=/usr/lib64 -lmqm_r -lpthread
```

Linux *Preparing COBOL programs in Linux*

Learn about preparing COBOL programs in Linux and preparing COBOL programs using IBM COBOL for Linux on x86 and Micro Focus COBOL.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

1. 32-bit COBOL copy books are installed in the following directory:

   ```
   MQ_INSTALLATION_PATH/inc/cobcpy32
   ```

   and symbolic links are created in:

   ```
   MQ_INSTALLATION_PATH/inc
   ```

2. On 64-bit platforms, 64 bit COBOL copy books are installed in the following directory:

   ```
   MQ_INSTALLATION_PATH/inc/cobcpy64
   ```

3. In the following examples set COBCPY to:

   ```
   MQ_INSTALLATION_PATH/inc/cobcpy32
   ```

   for 32-bit applications, and:

   ```
   MQ_INSTALLATION_PATH/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 IBM COBOL for Linux on x86

Sample COBOL programs are supplied with IBM MQ. To compile such a program, enter the appropriate command from the following list:

32-bit non-threaded server application

```
cob2 -o amq0put0 amq0put0.cbl -q"BINARY(BE)" -q"FLOAT(BE)" -q"UTF16(BE)"
-L${MQ_INSTALLATION_PATH}/lib -lmqmcb
-ICOBCPY_VALUE
```

32-bit non-threaded client application

```
cob2 -o amq0put0 amq0put0.cbl -q"BINARY(BE)" -q"FLOAT(BE)" -q"UTF16(BE)"
-L${MQ_INSTALLATION_PATH}/lib -lmqicb
-ICOBCPY_VALUE
```

32-bit threaded server application

```
cob2_r -o amq0put0 amq0put0.cbl -q"BINARY(BE)" -q"FLOAT(BE)" -q"UTF16(BE)" -qTHREAD
-L${MQ_INSTALLATION_PATH}/lib -lmqmcb_r
-ICOBCPY_VALUE
```

32-bit threaded client application

```
cob2_r -o amq0put0 amq0put0.cbl -q"BINARY(BE)" -q"FLOAT(BE)" -q"UTF16(BE)" -qTHREAD
-L${MQ_INSTALLATION_PATH}/lib -lmqicb_r
-ICOBCPY_VALUE
```

Preparing COBOL programs using Micro Focus COBOL

See the Micro Focus COBOL documentation for a description of the environment variables that you need.

Set environment variables before compiling your program:

```
export COBCPY=COBCPY_VALUE
export LIB=${MQ_INSTALLATION_PATH} lib:$LIB
```

Compile a 32-bit COBOL program (where supported) using Micro Focus COBOL, where `amqsput` is a sample program:

```
cob32 -xvP amqsput.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqmcb Server for COBOL
cob32 -xvP amqsput.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqicb Client for COBOL
cob32 -xtvP amqsput.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqmcb_r Threaded Server for COBOL
cob32 -xtvP amqsput.cbl -L${MQ_INSTALLATION_PATH}/lib -lmqicb_r Threaded Client for COBOL
```

Compile a 64-bit COBOL program using Micro Focus COBOL, where `amqsput` is a sample program:

```
cob64 -xvP amqsput.cbl -LMQ_INSTALLATION_PATH/lib64 -lmqmcb Server for COBOL
cob64 -xvP amqsput.cbl -LMQ_INSTALLATION_PATH/lib64 -lmqicb Client for COBOL
cob64 -xtvP amqsput.cbl -LMQ_INSTALLATION_PATH/lib64 -lmqmcb_r Threaded Server for COBOL
cob64 -xtvP amqsput.cbl -LMQ_INSTALLATION_PATH/lib64 -lmqicb_r Threaded Client for COBOL
```

Windows Building your procedural application on Windows

The Windows systems publications describe how to build executable applications from the programs that you write.

This topic describes the additional tasks, and the changes to the standard tasks, that you must perform when building IBM MQ for Windows applications to run under Windows systems. C, C++, COBOL, and Visual Basic programming languages are supported. For information about preparing your C++ programs, see Using C++.

The tasks that you must perform to create an executable application using IBM 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 IBM MQ for Windows

include files for the language that you are using. Make yourself familiar with the contents of these files. See “IBM MQ data definition files” on page 693 for a full description.

Windows *Building 64-bit applications on Windows*

Both 32-bit and 64-bit applications are supported on IBM MQ for Windows. The IBM 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 IBM MQ libraries are supplied in the following locations:

Table 148. Location of IBM MQ libraries

Library version	Directory containing library files
32-bit	*MQ_INSTALLATION_PATH* \Tools\Lib
64-bit	*MQ_INSTALLATION_PATH* \Tools\Lib64

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

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 *MQ_INSTALLATION_PATH* \Tools\Lib64. Ensure that the LIB environment variable is not set to look in the folder containing the 32-bit libraries.

Windows *Preparing C programs in Windows*

Work in your typical Windows environment; IBM MQ for Windows requires nothing special.

For further information about programming 64-bit applications see Coding standards on 64-bit platforms.

- Link your programs with the appropriate libraries provided by IBM MQ:

Library file	Program/exit type
MQ_INSTALLATION_PATH \Tools\Lib\mqm.lib	server for 32-bit C
MQ_INSTALLATION_PATH \Tools\Lib\mqic.lib	client for 32-bit C
MQ_INSTALLATION_PATH \Tools\Lib\mqicxa.lib	client for 32-bit C with transaction co-ordination
MQ_INSTALLATION_PATH \Tools\Lib64\mqm.lib	server for 64-bit C
MQ_INSTALLATION_PATH \Tools\Lib64\mqic.lib	client for 64-bit C

Library file	Program/exit type
MQ_INSTALLATION_PATH \Tools\Lib64\mqicxa.lib	client for 64-bit C with transaction co-ordination

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

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 -Feamqsget.exe ${MQ_INSTALLATION_PATH}\Tools\Lib\mqm.lib
```

For 64-bit applications:

```
cl -MD amqsget0.c -Feamqsget.exe ${MQ_INSTALLATION_PATH}\Tools\Lib64\mqm.lib
```

Note:

- If you are writing an installable service (see the Administering IBM MQ 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 IBM 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 IBM MQ for Windows to IBM MQ for AIX or Linux systems, where use of lowercase is necessary.

Preparing CICS and Transaction Server programs

Sample C source for a CICS IBM 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 code on one line):

   ```
   set CICS_IBMC_FLAGS=-I${MQ_INSTALLATION_PATH}\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
   ```

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

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 Administering IBM MQ.

Windows *Preparing COBOL programs in Windows*

Use this information to learn to prepare COBOL programs in Windows, and preparing CICS and Transaction Server programs.

1. The 32-bit COBOL copy books are installed in the following directory: *MQ_INSTALLATION_PATH* \Tools\cobol\CopyBook.

2. The 64-bit COBOL copy books are installed in the following directory: *MQ_INSTALLATION_PATH* \Tools\cobol\CopyBook64
3. In the following examples set CopyBook to:

   ```
   CopyBook
   ```

   for 32-bit applications, and:

   ```
   CopyBook64
   ```

   for 64-bit applications.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

To prepare COBOL programs on Windows systems, link your program to one of the following libraries provided by IBM MQ:

Library file	Program or exit type
MQ_INSTALLATION_PATH \Tools\Lib\mqmcb	32-bit server for Micro Focus COBOL
MQ_INSTALLATION_PATH \Tools\Lib\mqiccb	32-bit client for Micro Focus COBOL
MQ_INSTALLATION_PATH \Tools\Lib64\mqmcb	64-bit server for Micro Focus COBOL
MQ_INSTALLATION_PATH \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 IBM MQ library.

Preparing COBOL programs using Micro Focus COBOL

Relink any existing 32-bit IBM 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 Micro Focus COBOL:

1. Set the COBCPY environment variable to point to the IBM MQ COBOL copybooks (enter the following code on one line):

   ```
   set COBCPY=${MQ_INSTALLATION_PATH}\
   Tools\Cobol\Copybook
   ```

2. Compile the program to give you an object file:

   ```
   cobol amq0put0 LITLINK
   ```

3. Link the object file to the run time system.
   - Set the LIB environment variable to point to the compiler COBOL libraries.
   - Link the object file for use on the IBM MQ server:

     ```
     cbllink amq0put0.obj mqmcb.lib
     ```

   - Or link the object file for use on the IBM 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 code on one line):

```
set CICS_IBMCOB_FLAGS=${MQ_INSTALLATION_PATH}\
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 mqmcb.lib
```

Windows *Preparing Visual Basic programs in Windows*

Information to consider when using Microsoft Visual Basic programs on Windows.

Deprecated From IBM MQ 9.0, support for Microsoft Visual Basic 6.0 is deprecated. IBM MQ classes for .NET are the recommended replacement technology. For more information, see Developing .NET applications.

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 1012 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 an IBM 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
```

Related concepts

"Coding in Visual Basic" on page 1012
Information to consider when coding IBM MQ programs in Microsoft Visual Basic. Visual Basic is supported only on Windows.

Related reference

"Linking Visual Basic applications with the IBM MQ MQI client code" on page 886
You can link Microsoft Visual Basic applications with the IBM MQ MQI client code on Windows.

Windows *SSPI security exit*

IBM MQ for Windows supplies a security exit for both the IBM MQ MQI client and the IBM MQ server. This is a channel-exit program that provides authentication for IBM 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" on page 1092.

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 149. Context initiators and their associated context acceptors

Context Initiator	Context Acceptor
MQCHT_CLNTCONN	MQCHT_SVRCONN
MQCHT_RECEIVER	MQCHT_SENDER

Table 149. Context initiators and their associated context acceptors (continued)	
Context Initiator	**Context Acceptor**
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 topic 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 “IBM MQ access control and Windows principals” on page 983). 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.

IBM MQ access control and Windows principals

The access control that IBM 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 IBM 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/
```

z/OS Building your procedural application on z/OS

The CICS, IMS, and z/OS publications describe how to build applications that run in these environments.

This collection of topics describes the additional tasks, and the changes to the standard tasks, that you must perform when building IBM MQ for z/OS applications for these environments. COBOL, C, C++, Assembler, and PL/I programming languages are supported. (For information about building C++ applications see Using C++.)

The tasks that you must perform to create an executable IBM 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 IBM MQ for z/OS data definition file for the language that you are using. Make yourself familiar with the contents of these files. See “IBM MQ data definition files” on page 693 for a full description.

Note

The name **thlqual** is the high-level qualifier of the installation library on z/OS.

z/OS *Preparing your program to run*

After you have written the program for your IBM 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 IBM 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), Linux or z/OS UNIX System Services) in which the application runs, and the structure of the data sets on your z/OS installation.

“Dynamically calling the IBM MQ stub” on page 990 describes an alternative method of making MQI calls in your programs so that you do not need to link-edit an IBM 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 IBM 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 IBM MQ for z/OS V7.

z/OS *Building 64 bit C applications*

In z/OS, 64 bit C applications are built using the LP64 compiler and binder options. The IBM MQ for z/OS *cmqc.h* header file recognizes when this option is provided to the compiler, and generates IBM MQ data types and structures appropriate for 64 bit operation.

C code built with this option must be built to use dynamic-link libraries (DLLs) appropriate for the coordination semantic required. To achieve this, you bind the compiled code with the appropriate side-deck defined in the following table:

Table 150. Side-deck name required for each coordination semantic	
Coordination	**Side-deck name**
Single phase commit MQI	CSQBMQ2X
Two phase commit with RRS coordination, using RRS verbs	CSQBRR2X
Two phase commit with RRS coordination, using MQI verbs	CSQBRI2X

Note: For 31-bit C applications you also set compiler options for the calling interface (either Language Environment or XPLINK), as described in "Building z/OS batch applications using 31-bit Language Environment or XPLINK" on page 986. For 64-bit C applications you do not specify the calling interface, because the only supported linkage is XPLINK.

Use the EDCQCB JCL procedure, supplied with *z/OS XL C/C++*, to build a single phase commit IBM MQ program as a batch job, as follows:

```
//PROCS  JCLLIB ORDER=CBC.SCCNPRC
//CLG EXEC EDCQCB,
// INFILE='thlqual.SCSQC37S(CSQ4BCG1)',   < MQ SAMPLES
// CPARM='RENT,SSCOM,DLL,LP64,LIST,NOMAR,NOSEQ',  < COMPILER OPTIONS
// LIBPRFX='CEE',            < PREFIX FOR LIBRARY DSN
// LNGPRFX='CBC',            < PREFIX FOR LANGUAGE DSN
// BPARM='MAP,XREF,RENT,DYNAM=DLL', < LINK EDIT OPTIONS
// OUTFILE='userid.LOAD(CSQ4BCG1),DISP=SHR'
//COMPILE.SYSLIB DD
//        DD
//        DD DISP=SHR,DSN=thlqual.SCSQC370
//BIND.SCSQDEFS DD DISP=SHR,DSN=thlqual.SCSQDEFS
//BIND.SYSIN DD *
INCLUDE SCSQDEFS(CSQBMQ2X)
NAME CSQ4BCG1
```

To build an RRS coordinated program in z/OS UNIX System Services, compile and link as follows:

```
cc -o mqsamp -W c,LP64,DLL -W l,DYNAM=DLL,LP64 -I"//'thlqual.SCSQC370'" "//'thlqual.SCSQDEFS(CSQBRR2X)'" mqsamp.c
```

z/OS *Building z/OS batch applications*

Learn how to build z/OS batch applications and the steps to consider when doing so.

To build an application for IBM 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 IBM 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" on page 985.
3. For PL/I applications, use the compiler option EXTRN(SHORT).
4. Link-edit the object code created in step "1" on page 985 (or step "2" on page 985 for a C application) to produce a load module. When you link-edit the code, you must include one of the IBM MQ for z/OS batch stub programs (CSQBSTUB or one of the RRS stub programs: CSQBRRSI or CSQBRSTB).

   CSQBSTUB
   single-phase commit provided by IBM MQ for z/OS

   CSQBRRSI
   two-phase commit provided by RRS using the MQI

CSQBRSTB
two-phase commit provided by RRS directly

Notes:

a. If you use CSQBRSTB, you must also link-edit your application with ATRSCSS from SYS1.CSSLIB. Figure 113 on page 986 and Figure 114 on page 986 show fragments of JCL to do this. The stubs are language-independent and are supplied in library **thlqual**.SCSQLOAD.

b. If your application runs under Language Environment, you should ensure you link-edit with the Language Environment DLL instead as described in "Building z/OS batch applications using 31-bit Language Environment or XPLINK" on page 986.

5. Store the load module in an application load library.

```
⋮
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING BATCH STUB
//*
//CSQSTUB   DD  DSN=++THLQUAL++.SCSQLOAD,DISP=SHR
//*
⋮
//SYSIN     DD *
  INCLUDE CSQSTUB(CSQBSTUB)
⋮
/*
```

Figure 113. 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=++THLQUAL++.SCSQLOAD,DISP=SHR
//CSSLIB    DD  DSN=SYS1.CSSLIB,DISP=SHR
//*
⋮
//SYSIN     DD *
INCLUDE CSQSTUB(CSQBRSTB)
INCLUDE CSSLIB(ATRSCSS)
⋮
/*
```

Figure 114. 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 a batch program from the z/OS UNIX System Services 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
```

z/OS *Building z/OS batch applications using 31-bit Language Environment or XPLINK*

IBM MQ for z/OS provides a set of dynamic link libraries (DLLs) that must be used when you link-edit your applications.

There are two variants of the libraries that allow the application to use one of the following calling interfaces:

- The 31-bit Language Environment calling interface.

- The 31-bit XPLINK calling interface. z/OS XPLINK is a high performance calling convention available for C applications. See XPLINK | NOXPLINK in the z/OS 2.2 documentation.

To use the DLLs, the application is bound or linked against so called *sidedecks*, instead of the stubs provided with earlier versions. The sidedecks are found in the SCSQDEFS library (instead of the SCSQLOAD library).

Table 151. Variants of dynamic link libraries

Commit	31-bit Language Environment DLL	31-bit XPLINK DLL	Equivalent stub name
1 phase commit MQI libraries	CSQBMQ1	CSQBMQ1X	CSQBSTUB
2 phase commit with RRS co-ordination using RRS transaction-control verbs	CSQBRR1	CSQBRR1X	CSQBRSTB
2 phase commit with RRS co-ordination using MQI transaction-control verbs	CSQBRI1	CSQBRI1X	CSQBRRSI

Note: All sidedecks contain a definition of the data conversion entry point, MQXCNVC, previously resolved by including CSQASTUB.

Common issues:

- The following message appears on the job log if your application uses asynchronous message consume (MQCB, MQCTL or MQSUB calls) and the previous DLL interface is not used:

  ```
  CSQB001E Language environment programs running in z/OS batch or z/OS UNIX System Services must
  use the DLL interface to IBM MQ
  ```

  Solution: Rebuild your application using sidedecks instead of stubs as detailed previously.

- At program build time, the following message appears

  ```
  IEW2469E The Attributes of a reference to MQAPI-NAME from section your-code do not match the
  attributes of
  the target symbol
  ```

  Reason: This means that you have compiled your XPLINK program with V701 (or later) version of cmqc.h, but are not binding with sidedecks.

  Solution: Change your program's build file to bind against the appropriate sidedeck from SCSQDEFS instead of a stub from SCSQLOAD

The following sample JCL demonstrates how you can compile and link-edit a C program to use the 31 bit Language Environment DLL calling interface:

```
//CLG EXEC EDCCB,
//  INFILE=MYPROGS.CPROGS(MYPROGRAM),
//  CPARM='OPTF(DD:OPTF)',
//  BPARM='XREF,MAP,DYNAM=DLL'           < LINKEDIT OPTIONS
//COMPILE.OPTF DD *
RENT,CHECKOUT(ALL),SSCOM,DEFINE(MVS),NOMARGINS,NOSEQ,DLL
SE(DD:SYSLIBV)
//COMPILE.SYSLIB DD
//               DD
//               DD DISP=SHR,DSN=hlq.SCSQC370
//COMPILE.SYSLIBV DD DISP=SHR,DSN=hlq.BASE.H
/*
//BIND.SYSOBJ DD DISP=SHR,DSN=CEE.SCEEOBJ
//            DD DISP=SHR,DSN=hlq.SCSQDEFS
//BIND.SYSLMOD DD DISP=SHR,DSN=hlq.LOAD(MYPROGAM)
//BIND.SYSIN  DD *
 ENTRY CEESTART
 INCLUDE SYSOBJ(CSQBMQ1)
```

```
 NAME MYPROGAM(R)
//
```

Note: The compilation uses the **DLL** option. The link-edit uses **DYNAM=DLL** option and the references the **CSQBMQ1** library.

The following sample JCL demonstrates how you can compile and link-edit a C program to use the 31 bit XPLINK DLL calling interface:

```
//CLG  EXEC EDCXCB,
//  INFILE=MYPROGS.CPROGS(MYPROGRAM),
//  CPARM='OPTF(DD:OPTF)',
//  BPARM='XREF,MAP,DYNAM=DLL'           < LINKEDIT OPTIONS
//COMPILE.OPTF DD *
RENT,CHECKOUT(ALL),SSCOM,DEFINE(MVS),NOMARGINS,NOSEQ,XPLINK,DLL
SE(DD:SYSLIBV)
//COMPILE.SYSLIB DD
//               DD
//               DD DISP=SHR,DSN=hlq.SCSQC370
//COMPILE.SYSLIBV DD DISP=SHR,DSN=hlq.BASE.H
/*
//BIND.SYSOBJ DD DISP=SHR,DSN=CEE.SCEEOBJ
//            DD DISP=SHR,DSN=hlq.SCSQDEFS
//BIND.SYSLMOD DD DISP=SHR,DSN=hlq.LOAD(MYPROGAM)
//BIND.SYSIN  DD *
 ENTRY CEESTART
 INCLUDE SYSOBJ(CSQBMQ1X)
 NAME MYPROGAM(R)
//
```

Note: The compilation uses the **XPLINK** and **DLL** options. The link-edit uses **DYNAM=DLL** option and references the **CSQBMQ1X** library.

Ensure that you add the compilation option DLL to each program in the module. Messages such as IEW2456E 9207 SYMBOL CSQ1BAK UNRESOLVED are an indication that you need to check that all of the programs have been compiled with the DLL option.

z/OS *Building CICS applications in z/OS*

Use this information when building CICS applications in z/OS.

To build an application for IBM 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.
  - For PL/I programs, use the compiler option EXTRN(SHORT).
  - For C applications, if the application is not using XPLINK, use the compiler option DEFINE(MQ_OS_LINKAGE=1).
- 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 z/OS, the *CICS Transaction Server for z/OS 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 IBM 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 IBM MQ for z/OS CICS stub program (CSQCSTUB). Figure 115 on page 989 shows fragments of JCL code to do this. The stub is language-independent and is supplied in library **thlqual**.SCSQLOAD.

```
⋮
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING CICS STUB
//*
//CSQSTUB   DD  DSN=++THLQUAL++.SCSQLOAD,DISP=SHR
//*
⋮
//LKED.SYSIN DD *
  INCLUDE CSQSTUB(CSQCSTUB)
  ⋮
/*
```

Figure 115. Fragments of JCL to link-edit the object module in the CICS environment

- For CICS versions later than CICS TS 3.2, or, if you want to use IBM MQ message property APIs, or IBM MQ APIs MQCB, MQCTL, MQSTAT, MQSUB or MQSUBR, you must linkedit your object code with the CICS supplied stub, DFHMQSTB and not the IBM MQ supplied CSQCSTUB. For more information about building IBM MQ programs for CICS, see API stub program to access IBM MQ MQI calls in the CICS product documentation.

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.

Before you run a CICS program, your system administrator must define it to CICS as an IBM MQ program and transaction, You can then run it in the typical way.

z/OS *Building IMS (BMP or MPP) applications*

Use this information when building IMS (BMP or MPP) applications.

If you are building batch DL/I programs, see "Building z/OS batch applications" on page 985. 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 IBM 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 989.
3. For PL/I programs, use the compiler option EXTRN(SHORT).
4. For a C application, if the application is not using XPLINK, use the compiler option DEFINE(MQ_OS_LINKAGE=1).
5. Link-edit the object code created in step "1" on page 989 (or step "2" on page 989 for a C/370 application) to produce a load module:
   a. Include the IMS language interface module (DFSLI000).
   b. Include the IBM MQ for z/OS IMS stub program (CSQQSTUB). Figure 116 on page 990 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 IBM MQ stub" on page 990.

6. Store the load module in an application load library.

```
⋮
//*
//* WEBSPHERE MQ FOR Z/OS LIBRARY CONTAINING IMS STUB
//*
//CSQSTUB   DD  DSN=thlqual.SCSQLOAD,DISP=SHR
//*
⋮
//LKED.SYSIN DD *
  INCLUDE CSQSTUB(CSQQSTUB)
  ⋮
/*
```

Figure 116. 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 an IBM MQ program and transaction: you can then run it in the typical way.

z/OS *Building z/OS UNIX System Services applications*

Use this information when building z/OS UNIX System Services applications.

To build a C application for IBM MQ for z/OS that runs under z/OS UNIX System Services, compile and link your application as follows:

```
cc -o mqsamp -W c,DLL -I "//' thlqual.SCSQC370'" mqsamp.c "//' thlqual.SCSQDEFS(CSQBMQ1)'"
```

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 z/OS UNIX System Services, and enter z/OS UNIX System Services 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 IBM MQ calls.

"Dynamically calling the IBM MQ stub" on page 990 describes an alternative method of making MQI calls in your programs so that you do not need to link-edit an IBM 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 IBM MQ for z/OS on which your program is running. For example, a program running on IBM WebSphere MQ for z/OS 7.1 must not be link-edited with a stub program supplied with IBM MQ for z/OS 8.0.

z/OS *Dynamically calling the IBM MQ stub*

Instead of link-editing the IBM 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 in the RRS environment. If your application program uses RRS to coordinate updates, see "RRS Considerations" on page 995.

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 IBM 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 Setting up the IMS adapter) is ahead of the data set SCSQLOAD in the STEPLIB concatenation of the region JCL.

Use the names shown in Table 152 on page 991 when you call the stub dynamically. In PL/I, only declare the call names used in your program.

Table 152. Call names for dynamic linking

MQI call	Batch (non-RRS) dynamic call names	CICS dynamic call names	IMS dynamic call names
MQBACK	CSQBBACK	not supported	Not supported
MQBUFMH	CSQBBFMH	CSQCBFMH [1]	MQBUFMH
MQCB	CSQBCB	CSQCCB [1]	Not supported
MQCLOSE	CSQBCLOS	CSQCCLOS	MQCLOSE
MQCMIT	CSQBCOMM	not supported	Not supported
MQCONN	CSQBCONN	CSQCCONN	MQCONN
MQCONNX	CSQBCONX	CSQCCONX	MQCONNX
MQCRTMH	CSQBCTMH	CSQCCTMH [1]	MQCRTMH
MQCTL	CSQBCTL	CSQCCTL [1]	Not supported
MQDISC	CSQBDISC	CSQCDISC	MQDISC
MQDLTMH	CSQBDTMH	CSQCDTMH [1]	MQDLTMH
MQDLTMP	CSQBDTMP	CSQCDTMP [1]	MQDLTMP
MQGET	CSQBGET	CSQCGET	MQGET
MQINQ	CSQBINQ	CSQCINQ	MQINQ
MQINQMP	CSQBIQMP	CSQCIQMP [1]	MQINQMP
MQMHBUF	CSQBMHBF	CSQCMHBF [1]	MQMHBUF
MQOPEN	CSQBOPEN	CSQCOPEN	MQOPEN
MQPUT	CSQBPUT	CSQCPUT	MQPUT
MQPUT1	CSQBPUT1	CSQCPUT1	MQPUT1
MQSET	CSQBSET	CSQCSET	MQSET
MQSETMP	CSQBSTMP	CSQCSTMP [1]	MQSETMP
MQSTAT	CSQBSTAT	CSQCSTAT [1]	MQSTAT
MQSUB	CSQBSUB	CSQCSUB [1]	MQSUB
MQSUBRQ	CSQBSUBR	CSQCSUBR [1]	MQSUBRQ

Note: 1. These API calls are available only when using CICS TS 3.2 or later and the CSQCSTUB shipped with CICS must be used. For CICS TS 3.2, APAR PK66866 must be applied. For CICS TS 4.1, APAR PK89844 must be applied.

For examples of how to use this technique, see the following figures:

- Batch and COBOL: see Figure 117 on page 992
- CICS and COBOL: see Figure 118 on page 992
- IMS and COBOL: see Figure 119 on page 993
- Batch and assembler: see Figure 120 on page 993
- CICS and assembler: see Figure 121 on page 993
- IMS and assembler: see Figure 122 on page 993
- Batch and C: Figure 123 on page 994
- CICS and C: see Figure 124 on page 994
- IMS and C: see Figure 125 on page 994
- Batch and PL/I: see Figure 126 on page 994
- IMS and PL/I: see Figure 127 on page 995

```
...
       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 117. 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 118. 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 compilation 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 119. Dynamic linking using COBOL in the IMS environment

```
...
         LOAD   EP=CSQBOPEN
...
         CALL (15),(HCONN,MQOD,OPTIONS,HOBJ,COMPCODE,REASON),VL
...
         DELETE EP=CSQBOPEN
...
```

Figure 120. Dynamic linking using assembly 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 121. Dynamic linking using assembly language in the CICS environment

```
...
         LOAD   EP=MQOPEN
...
         CALL (15),(HCONN,MQOD,OPTIONS,HOBJ,COMPCODE,REASON),VL
...
         DELETE EP=MQOPEN
...
```

Figure 122. Dynamic linking using assembly 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 123. 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 124. 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 125. 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 126. 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 127. Dynamic linking using PL/I in the IMS environment

z/OS *RRS Considerations*

Consider using this information if your application program uses RRS to coordinate updates.

IBM MQ provides two different stubs for batch programs which need RRS coordination - see "The RRS batch adapter" on page 859. The difference in behavior of later 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 IBM MQ was done by using 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.
```

z/OS *Debugging your programs*

Use this information to learn about debugging TSO and CICS programs, and an insight into CICS trace.

The main aids to debugging IBM 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:

- IBM MQ for z/OS messages, completion, and reason codes for IBM MQ for z/OS
- Messages and reason codes for all other IBM MQ platforms

This topic also suggests other debugging tools to use in particular environments.

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

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 z/OS 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 z/OS 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 153 on page 996.

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 153. CICS adapter trace entries

Name	Description	Trace sequence	Trace data
CSQCABNT	Abnormal termination	Before issuing END_THREAD ABNORMAL to IBM 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 IBM MQ for z/OS log print utility.)
CSQCBACK	Syncpoint backout	Before issuing BACKOUT to IBM 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 IBM 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 an explicit syncpoint request from the application.	Unit of work information.

Table 153. CICS adapter trace entries (continued)

Name	Description	Trace sequence	Trace data
CSQCEXER	Execute resolve	Before issuing EXECUTE_RESOLVE to IBM 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 IBM 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 IBM MQ for z/OS.	Object handle.
CSQCLOSH	CLOSE handle	Before issuing MQCLOSE to IBM MQ for z/OS.	Object handle.
CSQCLOST	Disposition lost	During the resynchronization process, CICS informs the adapter that it has been restarted 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.
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 IBM 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.

Table 153. CICS adapter trace entries (continued)

Name	Description	Trace sequence	Trace data
CSQCOPNH	OPEN handle	After successful return from MQOPEN.	Object handle.
CSQCOPNO	OPEN object	Before issuing MQOPEN to IBM MQ for z/OS.	Object name.
CSQCPMGD	PUT message data	Before issuing MQPUT to IBM MQ for z/OS.	Up to 40 bytes of the message data.
CSQCPMGH	PUT message handle	Before issuing MQPUT to IBM MQ for z/OS.	Object handle.
CSQCPMGI	PUT message ID	After successful MQPUT from IBM MQ for z/OS.	Message ID and correlation ID of the message.
CSQCPREP	Syncpoint prepare	Before issuing PREPARE to IBM 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 IBM 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 IBM MQ for z/OS.	Object name.
CSQCRBAK	Resolved backout	Before issuing RESOLVE_ROLLBACK to IBM MQ for z/OS.	Unit of work information.
CSQCRCMT	Resolved commit	Before issuing RESOLVE_COMMIT to IBM MQ for z/OS.	Unit of work information.
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 z/OS 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 IBM MQ for z/OS.	Object handle.
CSQCTASE	IBM use only		

Table 153. CICS adapter trace entries (continued)			
Name	**Description**	**Trace sequence**	**Trace data**
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		

Handling procedural program errors

This information explains errors associated with your applications 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*.

Locally determined errors

Information about locally determined errors which include: failure on an MQI call, system interruptions, and messages containing incorrect data.

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 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 Return codes. For more detailed information, including ideas for corrective action, see:

- z/OS IBM MQ for z/OS messages, completion, and reason codes for IBM MQ for z/OS
- Messages and reason codes for all other IBM MQ platforms

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 sync points using:

- The IBM MQ for z/OS MQCMIT and MQBACK calls (see "Committing and backing out units of work" on page 823), or
- The z/OS Transaction Management and Recoverable Resource Manager Services (RRS) to provide two-phase sync point support. RRS allows you to update both IBM MQ and other RRS-enabled product resources, such as Db2 stored procedure resources, within a single logical unit of work. For information about RRS sync point support see "Transaction management and recoverable resource manager services" on page 827.

IBM i

You can make your MQPUT and MQGET calls within global units of work that are managed by IBM i commitment control. You can declare sync points by using the native IBM i COMMIT and ROLLBACK commands or the language-specific commands. Local units of work are managed by IBM MQ using the MQCMIT and MQBACK calls.

AIX, Linux, and Windows systems

In these environments, you can make your MQPUT and MQGET calls in the usual way, but you must declare sync points by using the MQCMIT and MQBACK calls (see "Committing and backing out units of work" on page 823). 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. ALW With IBM MQ on AIX, Linux, and 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 about log files on AIX, Linux, and Windows, see Administering IBM MQ. z/OS For z/OS see Planning on z/OS.

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. Exit the application or, alternatively, on IBM i IBM MQ for IBM i, AIX, Linux, and Windows systems, 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 with backout counts that are increasing over time are being repeatedly rejected; design your application so that it analyzes the reasons for this and handles such messages accordingly.

z/OS On IBM 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 IBM MQ for IBM i IBM i, AIX, Linux, and Windows systems, the backout count always survives the queue manager restarts.

z/OS Also, on IBM 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 771 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. See Feedback 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. See Structure of the report field for the options available.

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 about 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 try putting a message again 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 about using your own exit programs, see "Channel-exit programs for messaging channels" on page 926.

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; see MQDLH - Dead-letter header. 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 IBM 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 "Dead-letter queue processing" on page 1003.

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
This information contains general-use programming interface information when using dead-letter queue processing.

Dead-letter queue processing depends on local system requirements, but consider the following things 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 IBM 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 (Encoding, CodedCharSetId, and Format).

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

- 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
  - 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 about how undelivered messages are handled in a distributed queuing environment, see What happens when a message cannot be delivered?.

Multicast programming

Use this information to learn about the IBM MQ Multicast programming tasks such as connecting to a queue manager and exception reporting.

IBM MQ Multicast was designed to be as transparent to the user as possible and yet still be compatible with existing applications. Defining a COMMINFO object and setting the TOPIC object's **MCAST** and **COMMINFO** parameters, means that existing IBM MQ applications do not require substantial rewriting to use multicast. However, there might be some limitations (see “Multicast and the MQI” on page 1004 for more information) and some security issues to consider (see Multicast security for more information).

Multicast and the MQI

Use this information to understand the major Message Queue Interface (MQI) concepts and how they relate to IBM MQ Multicast.

Multicast subscriptions are nondurable; because there are no physical queues involved, there is nowhere to store the offline messages that are created by durable subscriptions.

After an application has subscribed to a multicast topic, it is given back an object handle which it can consume or MQGET from, as if it were a handle to a queue. This means that only managed multicast subscriptions (subscriptions created with MQSO_MANAGED) are supported, that is; it is not possible to make a subscription and 'point' the messages at a queue. This means that messages must be consumed from the object handle returned on the subscription call. On the client, the messages are stored in a message buffer until they are consumed by the client; see MessageBuffer stanza of the client configuration file for more information. If the client does not keep up with the publishing rate, the messages are discarded as required, with the oldest messages discarded first.

It is normally an administration decision whether an application uses Multicast or not, specified by setting the MCAST attribute of a TOPIC object. If a publishing application must ensure that multicast is not used, it can use the `MQOO_NO_MULTICAST` option. Similarly, a subscribing application can ensure that multicast is not used by subscribing with the `MQSO_NO_MULTICAST` option.

IBM MQ Multicast supports the use of message selectors. A selector is used by an application to register its interest in only those messages with properties that satisfy the SQL92 query that the selection string represents. For more information about message selectors, see “Selectors” on page 30.

The following table lists all the major MQI concepts and how they relate to Multicast:

Table 154. MQI concepts and how they relate to multicast

MQI Concept	Action when tried using multicast	Reason code
Putting a zero length message	Rejected	2005 (07D5) (RC2005): MQRC_BUFFER_LENGTH_ERROR
Grouping	Rejected	2046 (07FE) (RC2046): MQRC_OPTIONS_ERROR

Table 154. MQI concepts and how they relate to multicast (continued)

MQI Concept	Action when tried using multicast	Reason code
Segmentation	Rejected	2443 (098B) (RC2443): MQRC_SEGMENTATION_NOT_ALLOWED
Distribution lists	Rejected	2154 (086A) (RC2154): MQRC_RECS_PRESENT_ERROR
MQINQ	Rejected for topics handles: MQINQ and MQSET of topics is not supported.	2038 (07F6) (RC2038): MQRC_NOT_OPEN_FOR_INQUIRE
MQINQ	Accepted for managed handle. Only `Current Depth` can be inquired.	• If the value is Current Depth, then there is no applicable reason code. • If the value is anything other than Current Depth, the reason code is 2067 (0813) (RC2067): MQRC_SELECTOR_ERROR.
MQSET	Rejected for all handles.	2040 (07F8) (RC2040): MQRC_NOT_OPEN_FOR_SET
Transactions (XA or not)	Rejected	2072 (0818) (RC2072): MQRC_SYNCPOINT_NOT_AVAILABLE
Message browse	Rejected	2036 (07F4) (RC2036): MQRC_NOT_OPEN_FOR_BROWSE
Lock messages	Rejected	2046 (07FE) (RC2046): MQRC_OPTIONS_ERROR
Browse with mark	Rejected	2036 (07F4) (RC2036): MQRC_NOT_OPEN_FOR_BROWSE
Pass context	Rejected	2046 (07FE) (RC2046): MQRC_OPTIONS_ERROR
MQPUT1	Rejected. It is invalid to try and MQPUT1 to a Multicast only topic.	2560 (0A00) (RC2560): MQRC_MULTICAST_ONLY
Durable subscription	Rejected if the topic is marked as "Multicast only", otherwise a non-Multicast subscription is made.	2436 (0984) (RC2436): MQRC_DURABILITY_NOT_ALLOWED
TopicString > 255	Rejected. If the topic string is greater than 255 characters, it is rejected in the client.	2425 (0979) (RC2425): MQRC_TOPIC_STRING_ERROR

Table 154. MQI concepts and how they relate to multicast (continued)		
MQI Concept	**Action when tried using multicast**	**Reason code**
Non-managed subscription made	Rejected if the topic is marked as "Multicast only", otherwise a non-Multicast subscription is made.	2046 (07FE) (RC2046): MQRC_OPTIONS_ERROR
MQPMO_NOT_OWN_SUBS	Rejected	2046 (07FE) (RC2046): MQRC_OPTIONS_ERROR

The following items expand on some of the MQI concepts from the previous table, and provides information on some of the MQI concepts that are not in the table:

Message persistence
For nondurable multicast subscribers, persistent messages from the publisher are delivered in an unrecoverable fashion.

Message truncation
Message truncation is supported, which means that it is possible for an application to:

1. Issue an MQGET.
2. Get MQRC_TRUNCATED_MSG_FAILED.
3. Allocate a larger buffer.
4. Reissue the MQGET to retrieve the message.

Subscription expiry
Subscription expiry is not supported. Any attempt to set an expiry is ignored.

High availability for multicast

Use this information to understand IBM MQ Multicast continuous peer-to-peer operation; although IBM MQ connects to an IBM MQ queue manager, messages do not flow through that queue manager.

Although a connection to a queue manager must be made in order to MQOPEN or MQSUB the multicast topic object, the messages themselves do not flow through the queue manager. Therefore, after the MQOPEN or MQSUB is completed on the multicast topic object, it is possible to continue transmitting multicast messages even if the connection to the queue manager has been lost. There are two modes of operation:

A normal connection is made to the queue manager
Multicast communication is possible while the connection to the queue manager exists. If the connection fails, the normal MQI rules are applied, for example; an MQPUT to the multicast object handle returns 2009 (07D9) (RC2009): MQRC_CONNECTION_BROKEN.

A reconnecting client connection is made to the queue manager
Multicast communication is possible even during the reconnection cycle. This means that even when the connection to the queue manager has been broken, the putting and consuming of multicast messages is not affected. The client attempts to reconnect to a queue manager, and if that reconnection fails, the connection handle becomes broken and all MQI calls, including multicast ones, fail. For more information, see: Automatic client reconnection

If any application explicitly issues an MQDISC, then all multicast subscriptions and object handles are closed.

Multicast continuous peer-to-peer operation

One of the advantages of peer-to-peer communication between the clients is that the messages do not need to flow through the queue manager; therefore if the connection to the queue manager breaks, message transfer continues. The following restrictions apply to the continuous message requirements of this mode:

- The connection must be made using one of the MQCNO_RECONNECT_* options for continuous operation. This process means that although the communications session might be broken, the actual connection handle is not broken, and is in the reconnecting state instead. If reconnection fails, the connection handle is now broken which prevents all further MQI calls.
- Only MQPUT, MQGET, MQINQ, and Async Consume are supported in this mode. Any MQOPEN, MQCLOSE, or MQDISC verbs require reconnection to the queue manager to complete.
- Status flows to the queue manager stop; any state in the queue manager might therefore be stale or missing. This means that the clients might be sending and receiving messages and there is no status known on the queue manager. For more information, see: Multicast application monitoring

Data conversion in the MQI for multicast messaging

Use this information to understand how data conversion works for IBM MQ Multicast messaging.

IBM MQ Multicast is a shared, connectionless protocol, and so it is not possible for each client to make specific requests for data conversion. Every client subscribed to the same multicast stream receives the same binary data; therefore, if IBM MQ data conversion is required, the conversion is performed locally at each client.

Data is converted on the client for IBM MQ Multicast traffic. If the **MQGMO_CONVERT** option is specified, data conversion is done as requested. User defined formats need the data conversion exit installed on the client; see “Writing data-conversion exits” on page 947 for information about which libraries are now in the client and server packages.

For information about administering data conversion, see Enabling data conversion for Multicast messaging.

For more information about data conversion, see Data conversion.

For more information about data conversion exits and `ClientExitPath`, see ClientExitPath stanza of the client configuration file.

Multicast exception reporting

Use this information to learn about IBM MQ Multicast event handlers and reporting IBM MQ Multicast exceptions.

IBM MQ Multicast assists with problem determination by calling the event handler to report multicast events which are reported using the standard IBM MQ event handler mechanism.

An individual Multicast event can result in more than one IBM MQ event being called because there might be multiple `MQHCONN` connection handles using the same multicast transmitter or receiver. However, each multicast exception causes only one event handler to be called per IBM MQ connection.

The IBM MQ `MQCBDO_EVENT_CALL` constant enables applications to register a callback to receive only IBM MQ events, and the `MQCBDO_MC_EVENT_CALL` enable applications to register a callback to receive only multicast events. If both constants are used, both types of event are received.

Requesting Multicast events

IBM MQ Multicast events use the `MQCBDO_MC_EVENT_CALL` constant in the `cbd.Options` field. The following example demonstrates how to request multicast events:

```
cbd.CallbackType     = MQCBT_EVENT_HANDLER;
cbd.Options          = MQCBDO_MC_EVENT_CALL;
```

```
cbd.CallbackFunction = EventHandler;
MQCB(Hcon,MQOP_REGISTER,&cbd,MQHO_UNUSABLE_HOBJ,NULL,NULL,&CompCode,&Reason);
```

When the MQCBDO_MC_EVENT_CALL option is specified for the cbd.Options field, the event handler is sent only IBM MQ Multicast events instead of connection level events. To request that both types of events are sent to the event handler, the application must specify the MQCBDO_EVENT_CALL constant in the cbd.Options field as well as the MQCBDO_MC_EVENT_CALL constant as shown in the following example:

```
cbd.CallbackType     = MQCBT_EVENT_HANDLER;
cbd.Options          = MQCBDO_EVENT_CALL | MQCBDO_MC_EVENT_CALL
cbd.CallbackFunction = EventHandler;
MQCB(Hcon,MQOP_REGISTER,&cbd,MQHO_UNUSABLE_HOBJ,NULL,NULL,&CompCode,&Reason);
```

If neither of these constants is used, only connection level events are sent to the event handler.

For more information about values for the Options field see Options (MQLONG).

Multicast event format

IBM MQ Multicast exceptions include some supporting information which is returned in the **Buffer** parameter of the callback function. The **Buffer** pointer points to an array of pointers and the MQCBC.DataLength field specifies the size, in bytes, of the array. The first element of the array always points to a short text description of the event. More parameters might be supplied depending on the type of event. The following table lists the exceptions:

Table 155. Multicast event code descriptions

Event code	Description	Additional data
MQMCEV_PACKET_LOSS	Unrecoverable packet loss	Number of lost packets
MQMCEV_HEARTBEAT_TIMEOUT	Long absence of heartbeat control packet	N/A
MQMCEV_VERSION_CONFLICT	Reception of newer protocol version packets	N/A
MQMCEV_RELIABILITY	Different reliability modes of the transmitter and the receiver	N/A
MQMCEV_CLOSED_TRANS	Topic transmission is closed by 1 source	N/A
MQMCEV_STREAM_ERROR	Error detected on stream	N/A
MQMCEV_NEW_SOURCE	A new source starts to transmit on the topic	Source structure
MQMCEV_RECEIVE_QUEUE_TRIMMED	Packets removed from PacketQ due to time or space expiration	Number of trimmed packets
MQMCEV_PACKET_LOSS_NACK_EXPIRE	Unrecoverable packet loss due to NACK expiration	Number of lost packets
MQMCEV_ACK_RETRIES_EXCEEDED	Packets removed from history after **max_ack_retries** was exceeded	Number of packets removed
MQMCEV_STREAM_SUSPEND_NACK	NACKs have been suspended on a stream accepted by this topic	Suspend stream ID Time in milliseconds that the stream is suspended for

Table 155. Multicast event code descriptions (continued)

Event code	Description	Additional data
MQMCEV_STREAM_RESUME_NACK	NACKs have been resumed after they have been suspended on a stream	Stream ID
MQMCEV_STREAM_EXPELLED	A stream accepted by this topic has been rejected due to an expel request	Stream ID
MQMCEV_FIRST_MESSAGE	First message from a source	Message number
MQMCEV_LATE_JOIN_FAILURE	Failed to start late join session	N/A
MQMCEV_MESSAGE_LOSS	Unrecoverable message loss	Number of lost messages
MQMCEV_SEND_PACKET_FAILURE	Multicast transmitter failed to send a multicast packet	N/A
MQMCEV_REPAIR_DELAY	Multicast receiver did not receive a repair packet for an outstanding NAK	N/A
MQMCEV_MEMORY_ALERT_ON	Receiver reception buffers are filling up	Buffer pool utilization percentage
MQMCEV_MEMORY_ALERT_OFF	Receiver reception buffers are down to normal	Buffer pool utilization percentage
MQMCEV_NACK_ALERT_ON	Receiver repair packet request rate reached high water mark	Current repair request rate in packets per second
MQMCEV_NACK_ALERT_OFF	Receiver repair packet request rate is down to normal	Current repair request rate in packets per second
MQMCEV_REPAIR_ALERT_ON	Transmitter repair packet send rate reached high water mark	N/A
MQMCEV_REPAIR_ALERT_OFF	Transmitter repair packet send rate is down to normal	N/A
MQMCEV_SHM_DEST_UNUSABLE	The Shared Memory region used by a transmitter topic destination has been detected to be unusable	N/A
MQMCEV_SHM_PORT_UNUSABLE	The Shared Memory port used by a receiver instance has been detected to be unusable	N/A
MQMCEV_CCT_GETTIME_FAILED	The get time from Coordinated Cluster Time failed	N/A
MQMCEV_DEST_INTERFACE_FAILURE	The network interface used by a transmitter topic destination has failed and a backup network interface is unavailable	
MQMCEV_DEST_INTERFACE_FAILOVER	The network interface used by a transmitter topic destination has failed and a successful failover to another Interface has been completed	

Table 155. Multicast event code descriptions (continued)		
Event code	**Description**	**Additional data**
MQMCEV_PORT_INTERFACE-FAILURE	The network interface used by a receiver rmmPort has failed and a backup network interface is unavailable (or has also failed)	RMM configuration
MQMCEV_PORT_INTERFACE_FAILOVER	The network interface used by a receiver rmmPort has failed and a successful failover to another Interface has been completed	RMM configuration

Coding in C

Note the information in the following sections when coding IBM 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 typically 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 data can be specified as the parameter on the function invocation.

Data types

All data types are defined with 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 code 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 with a length, excluding the implied null, that exactly matches the defined length of the field. In the following examples, the symbol ¬ represents a single 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 typically 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
```

Windows Coding in Visual Basic

Information to consider when coding IBM MQ programs in Microsoft Visual Basic. Visual Basic is supported only on Windows.

Note:

Stabilized From IBM WebSphere MQ 7.0, outside the .NET environment, support for Visual Basic (VB) has been stabilized at the IBM WebSphere MQ 6.0 level. Most new function added to IBM WebSphere MQ

7.0 or later is not available to VB applications. If you are programming in VB.NET, use the IBM MQ classes for .NET. For more information, see Developing .NET applications.

Deprecated From IBM MQ 9.0, support for Microsoft Visual Basic 6.0 is deprecated. IBM MQ classes for .NET are the recommended replacement technology.

To avoid unintended translation of binary data passing between Visual Basic and IBM 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 IBM MQ structures. For example, for the MQMD (message descriptor) structure, MsgId (message identifier) is defined as MQBYTE24.

Visual Basic does not have a pointer data type, so references to other IBM 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. IBM 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 typical 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 data type rather than of MQCNO data type. This allows the function to accept either the MQCNO or the MQCNOCD structure. This function is declared in the main header file CMQB.

Related concepts

"Preparing Visual Basic programs in Windows" on page 981
Information to consider when using Microsoft Visual Basic programs on Windows.

Related reference

"Linking Visual Basic applications with the IBM MQ MQI client code" on page 886
You can link Microsoft Visual Basic applications with the IBM MQ MQI client code on Windows.

Coding in COBOL

Note the information in the following section when coding IBM MQ programs in COBOL.

Named constants

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.

Ensuring structure alignment

Care should be taken to ensure IBM MQ structures that are passed on to start on the MQ call's must be aligned on word boundaries. A word boundary is 4 bytes for 32-bit processes, 8 bytes for 64 bit processes and 16 bytes for 128 bit processes (IBM i).

Where possible, place all IBM MQ structures together so they are all boundary aligned.

Coding in System/390 assembler language (Message queue interface)

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

Names

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 IBM 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 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 Call descriptions 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 with 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

IBM 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” on page 1016. For more examples of how to do this, see the assembler language sample programs as shipped with IBM MQ.

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

For information on writing reenterable programs, see *z/OS MVS Application Development Guide: Assembler Language Programs*.

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 previous example is reenterable assembler-language code where PARMAREA is an area in the working storage that you specified.

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.

IBM i Coding IBM MQ programs in RPG (IBM i only)

In the IBM MQ documentation, the parameters of calls, the names of data types, the fields of structures, and the names of constants are all 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 IBM i Application Programming Reference (ILE/RPG).

Coding in PL/I (z/OS only)

Useful information when coding for IBM MQ 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';
```

Using the IBM MQ sample procedural programs

These sample programs are written in procedural languages, and demonstrate typical uses of the Message Queue Interface (MQI). IBM MQ programs on different platforms.

About this task

There are two sets of samples:

- Multi Sample programs for Multiplatforms.
- z/OS Sample programs for z/OS.

Procedure

- Use the following links to find out more about the sample programs:
  - Multi "Using the sample programs on Multiplatforms" on page 1018
  - z/OS "Using the sample programs for z/OS" on page 1118

Related concepts

"Application development concepts" on page 6
You can use a choice of procedural or object-oriented languages to write IBM MQ applications. Before you start to design and write your IBM MQ applications, familiarize yourself with the basic IBM MQ concepts.

"Developing applications for IBM MQ" on page 5
You can develop applications to send and receive messages, and to manage your queue managers and related resources. IBM MQ supports applications written in many different languages and frameworks.

"Design considerations for IBM MQ applications" on page 47
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 IBM MQ.

"Writing a procedural application for queuing" on page 696
Use this information to learn about writing queuing applications, connecting to and disconnecting from a queue manager, publish/subscribe, and opening and closing objects.

"Writing client procedural applications" on page 879
What you need to know to write client applications on IBM MQ using a procedural language.

"Writing publish/subscribe applications" on page 780
Start writing publish/subscribe IBM MQ applications.

"Building a procedural application" on page 962
You can write an IBM MQ application in one of several procedural languages, and run the application on several different platforms.

"Handling procedural program errors" on page 999
This information explains errors associated with your applications MQI calls either when it makes a call, or when its message is delivered to its final destination.

Multi Using the sample programs on Multiplatforms

These sample procedural programs are delivered with the product. The samples are written in C and COBOL, and demonstrate typical uses of the Message Queue Interface (MQI).

About this task

The samples are not intended to demonstrate general programming techniques, so some error checking that you might want to include in a production program is omitted.

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.

IBM i For RPG programming, see IBM i Application Programming Reference (ILE/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

The eighth character of the executable indicates whether the sample runs in local binding mode or client mode. If there is no eighth character, then the sample runs in local bindings mode. If the eighth character is 'c' then the sample runs in client mode.

Before you can run the sample applications, you must first create and configure a queue manager. To set up the queue manager to accept client connections, see "Configuring a queue manager to accept client connections on Multiplatforms" on page 1028.

Procedure

- Use the following links to find out more about the sample programs:
  - "Features demonstrated in the sample programs on Multiplatforms" on page 1020
  - "Preparing and running the sample programs" on page 1028
  - "The API exit sample program" on page 1035
  - "The Asynchronous consumption sample program" on page 1036
  - "The Asynchronous Put sample program" on page 1037
  - "The Browse sample programs" on page 1038
  - "The Browser sample program" on page 1039
  - "The CICS transaction sample" on page 1040
  - "The Connect sample program" on page 1041
  - "The Data-Conversion sample program" on page 1042
  - "Database coordination samples" on page 1042
  - "Dead-letter queue handler sample" on page 1049
  - "The Distribution List sample program" on page 1049
  - "The Echo sample programs" on page 1050
  - "The Get sample programs" on page 1051
  - "High availability sample programs" on page 1052
  - "The Inquire sample programs" on page 1056
  - "The Inquire Properties of a Message Handle sample program" on page 1057
  - "The Publish/Subscribe sample programs" on page 1057
  - "The Publish Exit sample program" on page 1062
  - "The Put sample programs" on page 1063
  - "The Reference Message sample programs" on page 1065
  - "The Request sample programs" on page 1072
  - "The Set sample programs" on page 1077
  - "The TLS sample program" on page 1078
  - "The Triggering sample programs" on page 1081

- “Using the TUXEDO samples on AIX, Linux, and Windows” on page 1083
- “Using the SSPI security exit on Windows” on page 1092
- “Running the samples using remote queues” on page 1093
- “The Cluster Queue Monitoring sample program (AMQSCLM)” on page 1093
- “Sample program for Connection Endpoint Lookup (CEPL)” on page 1103

Related concepts

“C++ sample programs” on page 511
Four sample programs are supplied, to demonstrate getting and putting messages.

Related tasks

“Using the sample programs for z/OS” on page 1118
The sample procedural applications that are delivered with IBM MQ for z/OS demonstrate typical uses of the Message Queue Interface (MQI).

Multi *Features demonstrated in the sample programs on Multiplatforms*

A collection of tables that show the techniques demonstrated by the IBM MQ sample programs.

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.

z/OS For the z/OS platform, see “Using the sample programs for z/OS” on page 1118.

Linux AIX *Samples for AIX and Linux systems*

The techniques demonstrated by the sample programs for IBM MQ for AIX or Linux.

See “Preparing and running sample programs on AIX and Linux” on page 1032 to find out where the sample programs for IBM MQ for AIX or Linux are stored.

Table 156 on page 1020 The table lists which C and COBOL source files are provided, and whether a server or client executable is included.

Table 156. Sample programs that demonstrate the use of the MQI (C and COBOL) on AIX and Linux.

A table with four columns. The first columns lists the techniques demonstrated by the samples. The second column lists the C samples and the third column lists the COBOL samples that demonstrate each of the techniques listed in the first column. The fourth column shows whether a server C executable is or is not included and the fifth column shows whether a client C executable is or is not included.

Technique	**C (source) (“1” on page 1023)**	**COBOL (source) (“2” on page 1023)**	**Server (C executable)**	**Client (C executable)**
Using the publish/subscribe interface	amqspuba amqssuba amqssbxa	no sample	amqspub amqssub amqssbx	no sample
Putting messages using the MQPUT call	amqsput0	amq0put0	amqsput	amqsputc
Putting a single message using the MQPUT1 call	amqsinqa amqsecha	amqminqx amqmechx amqiinqx amqiechx	amqsinq amqsech	amqsechc
Putting messages to a distribution list (“3” on page 1023)	amqsptl0	amq0ptl0.cbl	amqsptl	amqsptlc
Replying to a request message	amqsinqa	amqminqx amqiinqx	amqsinq	no sample

Table 156. Sample programs that demonstrate the use of the MQI (C and COBOL) on AIX and Linux.

A table with four columns. The first columns lists the techniques demonstrated by the samples. The second column lists the C samples and the third column lists the COBOL samples that demonstrate each of the techniques listed in the first column. The fourth column shows whether a server C executable is or is not included and the fifth column shows whether a client C executable is or is not included.

(continued)

Technique	**C (source) ("1" on page 1023)**	**COBOL (source) ("2" on page 1023)**	**Server (C executable)**	**Client (C executable)**
Getting messages using browse (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 ("3" on page 1023)	amqsprma	no sample	amqsprm	amqsprmc
Getting Reference Messages from a queue ("3" on page 1023)	amqsgrma	no sample	amqsgrm	amqsgrmc
Reference Message channel exit ("3" on page 1023)	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	amqsinq	amqsinqc
Using an exclusive input queue	amqstrg0	amq0req0	amqstrg	amqstrgc
Using the MQINQ call	amqsinqa	amqminqx amqiinqx	amqsinq	no sample
Using the MQSET call	amqsseta	amqmsetx amqisetx	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	("4" on page 1023)	no sample	no sample	no sample
IBM MQ (coordinating XA-compliant database managers) accessing a single database using SQL	amqsxas0.sqc Db2 amqsxas0.ec Informix	amq0xas0.sqb	no sample	no sample

Table 156. Sample programs that demonstrate the use of the MQI (C and COBOL) on AIX and Linux.

A table with four columns. The first columns lists the techniques demonstrated by the samples. The second column lists the C samples and the third column lists the COBOL samples that demonstrate each of the techniques listed in the first column. The fourth column shows whether a server C executable is or is not included and the fifth column shows whether a client C executable is or is not included.

(continued)

Technique	**C (source) ("1" on page 1023)**	**COBOL (source) ("2" on page 1023)**	**Server (C executable)**	**Client (C executable)**
IBM 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 ("5" on page 1023)	amqscic0.ccs	no sample	amqscic0	no sample
Encina transaction ("3" on page 1023)	amqsxae0	no sample	amqsxae0	no sample
TUXEDO transaction to put messages "6" on page 1023)	amqstxpx	no sample	no sample	no sample
TUXEDO transaction to get messages ("6" on page 1023)	amqstxgx	no sample	no sample	no sample
Server for TUXEDO ("6" on page 1023)	amqstxsx	no sample	no sample	no sample
Dead-letter queue handler	Directory `./tools/c/Samples/dlq` ("7" on page 1023)	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	no sample	amqsaxe	no sample
Cluster workload balancing exit	amqswlm0	no sample	amqswlm	no sample
Putting messages asynchronously and getting status using the MQSTAT call	amqsapt0	no sample	amqsapt	amqsaptc
Reconnectable clients	amqsphac amqsghac amqsmhac	no sample	not applicable	amqsphac amqsghac amqsmhac
Using message consumers to asynchronously consume messages from multiple queues	amqscbf0	no sample	amqscbf	amqscbfc
Specifying TLS connection information on MQCONNX	amqssslc	no sample	not applicable	amqssslc

Notes:

1. The executable version of the IBM MQ MQI client samples share the same source as the samples that run in a server environment.
2. 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. AIX Supported on IBM MQ for AIX only.
4. AIX On IBM MQ for AIX this program is called `amqsvfc0.c`
5. AIX CICS is supported by IBM MQ for AIX only.
6. Linux TUXEDO is not supported by IBM MQ for Linux on System p.
7. The source for the dead-letter queue handler consists of several files and is provided in a separate directory.

For detailed information about support for AIX and Linux systems, see System Requirements for IBM MQ.

Windows *Samples for IBM MQ for Windows*

The techniques demonstrated by the sample programs for IBM MQ for Windows.

Table 157 on page 1023 lists which C and COBOL source files are provided, and whether a server or client executable is included.

Table 157. IBM MQ for Windows sample programs demonstrating use of the MQI (C and COBOL)

Technique	**C (source)**	**COBOL (source)**	**Server (C executable)**	**Client (C executable)**
Using the publish/subscribe interface	amqspuba amqssuba amqssbxa	no sample	amqspub amqssub amqssbx	no sample
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

Table 157. IBM MQ for Windows sample programs demonstrating use of the MQI (C and COBOL) (continued)

Technique	C (source)	COBOL (source)	Server (C executable)	Client (C executable)
Using an exclusive input queue	amqstrg0	amq0req0	amqstrg	amqstrgc
Using the MQINQ call	amqsinqa	amqminq2 amqiinq2	amqsinq	amqsinqc
Using the MQSET call	amqsseta	amqmset2 amqiset2	amqsset	amqssetc
Using the MQINQMP call	amqsiqma	no sample	no sample	no sample
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
IBM MQ (coordinating XA-compliant database managers) accessing a single database using SQL	amqsxas0.sqc Db2 amqsxas0.ec Informix	amq0xas0.sqb	no sample	no sample
IBM MQ (coordinating XA-compliant database managers) accessing two databases using SQL	amqsxag0.c amqsxab0.sqc Db2 amqsxaf0.sqc Db2	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	Directory `./tools/c/Samples/dlq` (“1” on page 1025)	no sample	amqsdlq	no sample
From an IBM MQ MQI client, putting a message	no sample	no sample	no sample	amqsputc
From an IBM MQ 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	no sample	amqsaxe	no sample
Cluster workload balancing	amqswlm0	no sample	amqswlm	no sample

Table 157. IBM MQ for Windows sample programs demonstrating use of the MQI (C and COBOL) (continued)				
Technique	**C (source)**	**COBOL (source)**	**Server (C executable)**	**Client (C executable)**
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
Reconnectable clients	amqsphac amqsghac amqsmhac	no sample	Not applicable	amqsphac amqsghac amqsmhac
Using message consumers to asynchronously consume messages from multiple queues	amqscbf0	no sample	amqscbf	amqscbfc
Specifying TLS connection information on MQCONNX	amqssslc	no sample	not applicable	amqssslc

Notes:

1. The source for the dead-letter queue handler consists of several files and is provided in a separate directory.

Windows *Visual Basic samples for IBM MQ for Windows*

The techniques demonstrated by the sample programs for IBM MQ on Windows systems.

Table 158 on page 1025 shows the techniques demonstrated by the IBM 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 IBM MQ server. To find out how to change the sample projects to work with the IBM MQ clients see “Preparing Visual Basic programs in Windows” on page 981.

Table 158. IBM 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

IBM i *Samples for IBM i*

The techniques demonstrated by the sample programs for IBM MQ on IBM i systems.

Table 159 on page 1026 shows the techniques demonstrated by the IBM MQ for IBM i sample programs. Some techniques occur in more than one sample program, but only one program is listed in the table.

Table 159. Sample programs demonstrating use of the MQI (C and COBOL) on IBM i

Technique	**C (source) ("1" on page 1027)**	**COBOL (source) ("2" on page 1027)**	**RPG (source) ("3" on page 1027)**	**Client (C executable) (4)**
Putting messages using the MQPUT call	AMQSPUT0	AMQ0PUT4	AMQ3PUT4	AMQSPUTC
Putting messages from a data file using the MQPUT call	AMQSPUT4	no sample	no sample	no sample
Putting a single message using the MQPUT1 call	AMQSINQ4, AMQSECH4	AMQ0INQ4, AMQ0ECH4	AMQ3INQ4, AMQ3ECH4	AMQSINQC, AMQSECHC
Putting messages to a distribution list	AMQSPTL4	no sample	no sample	AMQSPTLC
Replying to a request message	AMQSINQ4	AMQ0INQ4	AMQ3INQ4	AMQSINQC
Getting messages (no wait)	AMQSGBR4	AMQ0GBR4	AMQ3GBR4	AMQSGBRC
Getting messages (wait with a time limit)	AMQSGET4	AMQ0GET4	AMQ3GET4	AMQSGETC
Getting messages (unlimited wait)	AMQSTRG4	no sample	AMQ3TRG4	AMQSTRGC
Getting messages (with data conversion)	AMQSECH4	AMQ0ECH4	AMQ3ECH4	AMQSECHC
Putting Reference Messages to a queue	AMQSPRM4	no sample	no sample	AMQSPRMC
Getting Reference Messages from a queue	AMQSGRM4	no sample	no sample	AMQSGRMC
Reference Message channel exit	AMQSQRM4, AMQSXRM4	no sample	no sample	no Sample
Message exit	AMQSCMX4	no sample	no sample	no Sample
Browsing first 49 characters of a message	AMQSGBR4	AMQ0GBR4	AMQ3GBR4	AMQSGBRC
Browsing complete messages	AMQSBCG4	no sample	no sample	AMQSBCGC
Using a shared input queue	AMQSINQ4	AMQ0INQ4	AMQ3INQ4	AMQSINQC
Using an exclusive input queue	AMQSREQ4	AMQ0REQ4	AMQ3REQ4	AMQSREQC
Using the MQINQ call	AMQSINQ4	AMQ0INQ4	AMQ3INQ4	AMQSINQC
Using the MQSET call	AMQSSET4	AMQ0SET4	AMQ3SET4	AMQSSETC
Using a reply-to queue	AMQSREQ4	AMQ0REQ4	AMQ3REQ4	AMQSREQC
Requesting message exceptions	AMQSREQ4	AMQ0REQ4	AMQ3REQ4	AMQSREQC
Accepting a truncated message	AMQSGBR4	AMQ0GBR4	AMQ3GBR4	AMQSGBRC
Using a resolved queue name	AMQSGBR4	AMQ0GBR4	AMQ3GBR4	AMQSGBRC
Triggering a process	AMQSTRG4	no sample	AMQ3TRG4	AMQSTRGC
Trigger server	AMQSERV4	no sample	AMQ3SRV4	no sample
Using a trigger server (including CICS transactions)	AMQSERV4	no sample	AMQ3SRV4	no sample

Table 159. Sample programs demonstrating use of the MQI (C and COBOL) on IBM i (continued)

Technique	C (source) ("1" on page 1027)	COBOL (source) ("2" on page 1027)	RPG (source) ("3" on page 1027)	Client (C executable) (4)
Using data conversion	AMQSVFC4	no sample	no sample	no sample
Using API exits	AMQSAXE0	no sample	no sample	no sample
Cluster workload balancing	AMQSWLM0	no sample	no sample	no sample
Putting messages asynchronously and getting status using the MQSTAT call	AMQSAPT0	no sample	no sample	AMQSAPTC
Using the publish/subscribe interface	AMQSPUBA, AMQSSUBA, AMQSSBXA	no sample	no sample	AMQSPUBC, AMQSSUBC, AMQSSBXC
Reconnectable clients (5)	AMQSPHAC, AMQSGHAC, AMQSMHAC	no sample	no sample	no sample
Using message consumers to asynchronously consume messages from multiple queues (5)	AMQSCBFO	no sample	no sample	no sample
Specifying TLS connection information on MQCONNX	AMQSSSLC	no sample	no sample	AMQSSSLC
Connecting to the queue manager using MQCONNX	AMQSCNXC	no sample	no sample	AMQSCNXC
Inquire properties of a message handle, using MQINQMP, from a message queue	AMQISQMA	no sample	no sample	AMQISQMC
Set properties of a message handle using MQSETMP and put it to a message queue	AMQSSQMA	no sample	no sample	AMQSSQMC

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 AMQ0 xxx 4, where xxx indicates the sample function.
3. Source for the RPG samples is in QMQMSAMP/QRPGLESRC. Members are named AMQ3 xxx 4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name has the suffix G.
4. The executable version of the IBM MQ MQI client samples share the same source as the samples that run in a server environment. Source for the samples in the client environment is same as the server. IBM MQ MQI client samples are linked with client library LIBMQIC and IBM MQ server samples are linked with server library LIBMQM.
5. If client executable for sample application of Reconnectable client and asynchronously consumer application has to be run, it has to be compiled and linked with threaded library LIBMQIC_R. Hence, it has to be run in threaded environment. Set the environment variable QIBM_MULTI_THREADED to 'Y' and run the application from qsh.

   See Setting up IBM MQ with Java and JMS for more information.

See "Preparing and running sample programs on IBM i" on page 1030 for more information.

In addition to these, the IBM MQ for IBM i 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 Administering IBM i . You could use the sample CL program amqsamp4 to create queues to use with the sample programs described in this topic.

Multi ***Preparing and running the sample programs***

After completing some initial preparation, you can then run the sample programs.

About this task

Before you run the sample programs, you must first create a queue manager and also create the queues that you need. You might also need to do some additional preparation, for example, if you want to run COBOL samples. After completing the necessary preparation, you can then run the sample programs.

Procedure

For information about how to prepare and run the sample programs, see the following topics:

- "Configuring a queue manager to accept client connections on Multiplatforms" on page 1028
- "Preparing and running sample programs on IBM i" on page 1030
- "Preparing and running sample programs on AIX and Linux" on page 1032
- "Preparing and running sample programs on Windows" on page 1033

Multi *Configuring a queue manager to accept client connections on Multiplatforms*

Before you can run the sample applications, you must first create a queue manager. You can then configure the queue manager to securely accept incoming connection requests from applications that are running in client mode.

Before you begin

Ensure the queue manager already exists and has been started. Determine whether channel authentication records are already enabled by issuing the MQSC command:

```
DISPLAY QMGR CHLAUTH
```

Important: This task expects that channel authentication records are enabled. If this is a queue manager used by other users and applications, changing this setting will affect all other users and applications. If your queue manager does not make use of channel authentication records then step 4 can be replaced with an alternate authentication method (for example, a security exit) which sets the MCAUSER to the *non-privileged-user-id* you will obtain in step "1" on page 1028.

You must know which channel name your application expects to use so that the application can be permitted to use the channel. You must also know which objects, for example queues or topics, your application expects to use so that your application can be permitted to use them.

About this task

This task creates a non-privileged user ID to be used for a client application which connects to the queue manager. Access is granted for the client application only to be able to use the channel it needs and the queue it needs by use of this user ID.

Procedure

1. Obtain a user ID on the system your queue manager is running on. For this task this user ID must not be a privileged administrative user. This user ID will be the authority under which the client connection will run on the queue manager.
2. Start a listener program with the following commands, where:

   qmgr-name is the name of your queue manager
   nnnn is your chosen port number

a) ALW

For AIX, Linux, and Windows systems:

```
runmqlsr -t tcp -m qmgr-name -p nnnn
```

b) IBM i

For IBM i:

```
STRMQMLSR MQMNAME(qmgr-name) PORT(nnnn)
```

3. If your application uses the SYSTEM.DEF.SVRCONN then this channel is already defined. If your application uses another channel, create it by issuing the following MQSC command:

```
DEFINE CHANNEL(' channel-name ') CHLTYPE(SVRCONN) TRPTYPE(TCP) +
DESCR('Channel for use by sample programs')
```

where *channel-name* is the name of your channel.

4. Create a channel authentication rule allowing only the IP address of your client system to use the channel by issuing the following MQSC command:

```
SET CHLAUTH(' channel-name ') TYPE(ADDRESSMAP) ADDRESS(' client-machine-IP-address ') +
MCAUSER(' non-privileged-user-id ')
```

where

channel-name is the name of your channel.

client-machine-IP-address is the IP address of your client system. If your sample client application is running on the same machine as the queue manager then use an IP address of '127.0.0.1' if your application is going to connect using 'localhost'. If several different client machines are going to connect in, you can use a pattern or a range instead of a single IP address. See Generic IP addresses for details.

non-privileged-user-id is the user ID you obtained in step “1” on page 1028

5. If your application uses the SYSTEM.DEFAULT.LOCAL.QUEUE then this queue is already defined. If your application uses another queue, create it by issuing the following MQSC command:

```
DEFINE QLOCAL(' queue-name ') DESCR('Queue for use by sample programs')
```

where *queue-name* is the name of your queue.

6. Grant access to connect to and inquire the queue manager by issuing the following MQSC command:

```
SET AUTHREC OBJTYPE(QMGR) PRINCIPAL(' non-privileged-user-id ') +
AUTHADD(CONNECT, INQ)
```

where *non-privileged-user-id* is the user ID you obtained in step “1” on page 1028 .

7. If your application is a point-to-point application, that is it makes use of queues, grant access to allow inquiring and the putting and getting messages using your queue by the user ID to be used, by issuing the following MQSC commands:

```
SET AUTHREC PROFILE(' queue-name ') OBJTYPE(QUEUE) +
PRINCIPAL(' non-privileged-user-id ') AUTHADD(PUT, GET, INQ, BROWSE)
```

where

queue-name is the name of your queue

non-privileged-user-id is the user ID you obtained in step “1” on page 1028

8. If your application is a publish/subscribe application, that is it makes use of topics, grant access to allow publishing and subscribing using your topic by the user ID to be used, by issuing the MQSC commands:

```
SET AUTHREC PROFILE('SYSTEM.BASE.TOPIC') OBJTYPE(TOPIC) +
PRINCIPAL(' non-privileged-user-id ') AUTHADD(PUB, SUB)
```

where

non-privileged-user-id is the user ID you obtained in step “1” on page 1028
This will give *non-privileged-user-id* access to any topic in the topic tree, alternatively, you can define a topic object using **DEFINE TOPIC** and grant accesses only to the part of the topic tree referenced by that topic object. See Controlling user access to topics for details.

What to do next

Your client application can now connect to the queue manager and put or get messages using the queue.

Related concepts

ALW Giving access to an IBM MQ object on AIX, Linux, and Windows

Related reference

SET CHLAUTH
DEFINE CHANNEL
DEFINE QLOCAL
SET AUTHREC
IBM i IBM MQ authorities on IBM i

IBM i *Preparing and running sample programs on IBM i*

Before you run the sample programs on IBM i, you must first create a queue manager and also create the queues that you need. If you want to run COBOL samples, you might need to do some additional preparation.

About this task

The source for the IBM MQ for IBM i sample programs is provided in library QMQMSAMP as members of QCSRC, QCLSRC, QCBLLESRC, and QRPGLESRC.

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 the AMQSAMP4 program is included in file QCLSRC in library QMQMSAMP. You can compile it by using the CRTCLPGM command.

To run the samples, either use the C executable versions, which are supplied in the library QMQM, or compile them in a similar way to any other IBM MQ application.

V 9.4.0 V 9.4.0 The following sample programs have authentication capabilities:

- `amqsbcg0.c`
- `amqsfhac.c`
- `amqsget0.c`
- `amqsghac.c`
- `amqsmhac.c`
- `amqsphac.c`
- `amqspuba.c`
- `amqsput0.c`
- `amqsssl c.c`
- `amqssuba.c`

The executable versions of these samples have authentication enabled. However, compiling the source versions with authentication enabled requires the compile flag **SAMPLE_AUTH_ENABLED** to be defined and the `amqsauth.c` source file to be compiled with the desired sample. For example:

- Creating the `amqsssl`c program without authentication enabled:

```
CRTCMOD MODULE(MYLIB/AMQSSSLC) SRCFILE(QMQMSAMP/QCSRC)
CRTPGM PGM(MYLIB/AMQSSSLC) MODULE(MYLIB/AMQSSSLC) BNDSRVPGM(QMQM/LIBMQIC)
```

- Creating the `amqsssl`c with authentication enabled:

```
CRTCMOD MODULE(MYLIB/AMQSSSLC) DEFINE('SAMPLE_AUTH_ENABLED') SRCFILE(QMQMSAMP/QCSRC)
CRTCMOD MODULE(MYLIB/AMQSAUTH) SRCFILE(QMQMSAMP/QCSRC)
CRTPGM PGM(MYLIB/AMQSSSLC_AUTH) MODULE(MYLIB/AMQSSSLC MYLIB/AMQSAUTH) BNDSRVPGM(QMQM/LIBMQIC)
```

Procedure

1. Create a queue manager and set up the default definitions.

   You must do this before you can run any of the sample programs. For more information about creating a queue manager, see Administering IBM MQ. For information about configuring a queue manager to securely accept incoming connection requests from applications that are running in client mode, see “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028.
2. To call one of the sample programs by using data from member PUT in file AMQSDATA of library QMQMSAMP, use a command like:

   ```
   CALL PGM(QMQM/AMQSPUT4) PARM('QMQMSAMP/AMQSDATA(PUT)')
   ```

   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
   ```
3. If you want to use the COBOL versions of the Inquire, Set, and Echo examples, change the process definitions before you run these samples.

   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 IBM i, you can use the **CHGMQMPRC** command (for details, see Change MQ Process (CHGMQMPRC)), or edit and run the **AMQSAMP4** command with the alternative definition.
4. Run the sample programs.

   For more information on the parameters that each of the samples expects, see the descriptions of the individual samples.

   Note: 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.

Related reference

“Samples for IBM i” on page 1026

The techniques demonstrated by the sample programs for IBM MQ on IBM i systems.

Linux AIX *Preparing and running sample programs on AIX and Linux*

Before you run the sample programs on AIX and Linux, you must first create a queue manager and also create the queues that you need. If you want to run COBOL samples, you might need to do some additional preparation.

About this task

The IBM MQ on AIX and Linux systems sample files are in the directories listed in Table 160 on page 1032 if the defaults were used at installation time.

Table 160. Where to find the samples for IBM MQ on AIX and Linux systems

Content	Directory
source files	*MQ_INSTALLATION_PATH*/samp
dead-letter queue handler source files	*MQ_INSTALLATION_PATH*/samp/dlq
executable files	*MQ_INSTALLATION_PATH*/samp/bin

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

The samples need a set of queues to work with. You can either use your own queues or run the sample MQSC file amqscos0.tst to create a set. 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.

V 9.4.0 V 9.4.0 The following sample programs have authentication capabilities:

- amqsbcg0.c
- amqsfhac.c
- amqsget0.c
- amqsghac.c
- amqsmhac.c
- amqsphac.c
- amqspuba.c
- amqsput0.c
- amqsсslc.c
- amqssuba.c

The executable versions of these samples have authentication enabled. However, compiling the source versions with authentication enabled requires the compile flag **SAMPLE_AUTH_ENABLED** to be defined and the amqsauth.c source file to be compiled with the desired sample. For example:

- Compiling amqsput0.c without authentication enabled:

```
gcc -m64 -I /opt/mqm/inc -L /opt/mqm/lib64 -lmqic -fsanitize=address -o /bin/amqsput
amqsput0.c
```

- Compiling amqsput0.c with authentication enabled:

```
gcc -m64 -I /opt/mqm/inc -L /opt/mqm/lib64 -lmqic -fsanitize=address -D SAMPLE_AUTH_ENABLED
-o /bin/amqsputc_auth amqsauth.c amqsput0.c
```

Procedure

1. Create a queue manager and set up the default definitions.

You must do this before you can run any of the sample programs. For more information about creating a queue manager, see Administering IBM MQ. For information about configuring a queue manager to securely accept incoming connection requests from applications that are running in client mode, see "Configuring a queue manager to accept client connections on Multiplatforms" on page 1028.

2. If you are not using your own queues, run the sample MQSC file amqscos0.tst to create a set of queues.

   To do this on AIX and Linux systems, enter:

   ```
   runmqsc QManagerName <amqscos0.tst > /tmp/sampobj.out
   ```

   Check the `sampobj.out` file to ensure that there are no errors.

3. If you want to use the COBOL versions of the Inquire, Set, and Echo examples, change the process definitions before you run these samples.

   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 AIX and Linux, 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 to run these samples.

4. Run the sample programs.

   To run a sample, enter its name 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*.

   For more information on the parameters that each of the samples expects, see the descriptions of the individual samples.

   Note: 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.

Related reference

"Samples for AIX and Linux systems" on page 1020
The techniques demonstrated by the sample programs for IBM MQ for AIX or Linux.

Windows *Preparing and running sample programs on Windows*

Before you run the sample programs on Windows, you must first create a queue manager and also create the queues that you need. If you want to run COBOL samples, you might need to do some additional preparation.

About this task

The IBM MQ for Windows sample files are in the directories listed in Table 161 on page 1033, if the defaults were used at installation time. The installation drive defaults to <c:>.

Table 161. Where to find the samples for IBM MQ for Windows

Content	Directory
C source code	*MQ_INSTALLATION_PATH*\Tools\C\Samples
Source code for dead-letter handler sample	*MQ_INSTALLATION_PATH*\Tools\C\Samples\DLQ

Table 161. Where to find the samples for IBM MQ for Windows (continued)	
Content	**Directory**
COBOL source code	*MQ_INSTALLATION_PATH*\Tools\Cobol\Samples
C executable files [1]	*MQ_INSTALLATION_PATH*\ Tools\C\Samples\Bin (32-bit versions) *MQ_INSTALLATION_PATH*\ Tools\C\Samples\Bin64 (64-bit versions)
Sample MQSC files	*MQ_INSTALLATION_PATH*\Tools\MQSC\Samples
Visual Basic source code	*MQ_INSTALLATION_PATH*\Tools\VB\SampVB6
.NET samples	*MQ_INSTALLATION_PATH*\Tools\dotnet\Samples

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

Note: 64-bit versions are available of some C executable file samples.

The samples need a set of queues to work with. You can either use your own queues or run the sample MQSC file amqscos0.tst to create a set of queues. To run the samples, either use the executable versions supplied or compile the source versions as you would any other IBM MQ for Windows applications.

V 9.4.0 V 9.4.0 The following sample programs have authentication capabilities:

- `amqsbcg0.c`
- `amqsfhac.c`
- `amqsget0.c`
- `amqsghac.c`
- `amqsmhac.c`
- `amqsphac.c`
- `amqspuba.c`
- `amqsput0.c`
- `amqsсslc.c`
- `amqssuba.c`

The executable versions of these samples have authentication enabled. However, compiling the source versions with authentication enabled requires the compile flag **SAMPLE_AUTH_ENABLED** to be defined and the `amqsauth.c` source file to be compiled with the desired sample. For example:

- Compiling `amqsput0.c` without authentication enabled:

```
CL amqsput0.c /link mqic.lib /OUT:Bin\amqsputc.exe
```

- Compiling `amqsput0.c` with authentication enabled:

```
CL /D SAMPLE_AUTH_ENABLED amqsauth.c amqsput0.c /link mqic.lib /OUT:Bin\amqsputc_auth.exe
```

Procedure

1. Create a queue manager and set up the default definitions.

   You must do this before you can run any of the sample programs. For more information about creating a queue manager, see Administering IBM MQ. For information about configuring a queue manager to securely accept incoming connection requests from applications that are running in client mode, see “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028.
2. If you are not using your own queues, run the sample MQSC file amqscos0.tst to create a set of queues.

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.

3. If you want to use the COBOL versions of the Inquire, Set, and Echo examples, change the process definitions before you run these samples.

   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, 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 to run these samples.

4. Run the sample programs.

   To run a sample, enter its name 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*.

   For more information on the parameters that each of the samples expects, see the descriptions of the individual samples.

   Note: 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.

Related reference

"Samples for IBM MQ for Windows" on page 1023
The techniques demonstrated by the sample programs for IBM MQ for Windows.

"Visual Basic samples for IBM MQ for Windows" on page 1025
The techniques demonstrated by the sample programs for IBM MQ on Windows systems.

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 "Writing and compiling API exits on Multiplatforms" on page 918.

Source
amqsaxe0.c

Binary
amqsaxe

Configuring for the sample exit

1. Add the following information to the ApiExitLocal stanza of the `qm.ini` file.

   Platforms other than Windows

   ```
   ApiExitLocal:
   Sequence=100
   Function=EntryPoint
   Module= MQ_INSTALLATION_PATH/samp/bin/amqsaxe
   Name=SampleApiExit
   ```

where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed.

Windows

```
ApiExitLocal:
Sequence=100
Function=EntryPoint
Module= MQ_INSTALLATION_PATH\Tools\c\Samples\bin\amqsaxe
Name=SampleApiExit
```

where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed.

2. Set the **MQAPI_TRACE_LOGFILE** environment variable

```
MQAPI_TRACE_LOGFILE=/tmp/MqiTrace
```

3. Run your application.

   Output files are created in the `/tmp` directory with names like: `MqiTrace.`*`pid`*`.`*`tid`*`.log`.

The Asynchronous consumption sample program

The amqscbf sample program demonstrates the use of MQCB and MQCTL to consume messages from multiple queues asynchronously.

amqscbf is provided as C source code, and a binary client and server executable on AIX, Linux, and Windows platforms.

The program is started from the command line and takes the following optional parameters:

```
Usage: [Options] Queue Name {queue_name}
  where Options are:
    -m Queue Manager Name
    -o Open options
    -r Reconnect Type
       d Reconnect Disabled
       r Reconnect
       m Reconnect Queue Manager
```

Provide more than one queue name to read messages from multiple queues (a maximum of ten queues are supported by the sample.)

Note: `Reconnect type` is only valid for client programs.

Example

The example shows amqscbf run as a server program reading one message from `QL1` and then being stopped.

Use IBM MQ Explorer to put a test message on `QL1`. Stop the program by pressing enter.

```
C:\>amqscbf QL1
Sample AMQSCBF0 start

Press enter to end
Message Call (9 Bytes) :
Message 1

Sample AMQSCBF0 end
```

What amqscbf demonstrates

The sample shows how to read messages from multiple queues in the order of their arrival. This would require a lot more code using synchronous MQGET. In the case of asynchronous consumption, no polling is required, and thread and storage management is performed by IBM MQ. A "real world" example would need to deal with errors; in the sample errors are written out to the console.

The sample code has the following steps,

1. Define the single message consumption callback function,

```
void MessageConsumer(MQHCONN      hConn,
                     MQMD    * pMsgDesc,
                     MQGMO   * pGetMsgOpts,
                     MQBYTE  * Buffer,
                     MQCBC   * pContext)
{ ... }
```

2. Connect to the queue manager,

```
MQCONNX(QMName,&cno,&Hcon,&CompCode,&CReason);
```

3. Open the input queues, and associate each one with the `MessageConsumer` callback function,

```
MQOPEN(Hcon,&od,O_options,&Hobj,&OpenCode,&Reason);
cbd.CallbackFunction = MessageConsumer;
MQCB(Hcon,MQOP_REGISTER,&cbd,Hobj,&md,&gmo,&CompCode,&Reason);
```

   `cbd.CallbackFunction` does not need to be set for each queue; it is an input-only field. But you could associate a different callback function with each queue.

4. Start consumption of the messages,

```
MQCTL(Hcon,MQOP_START,&ctlo,&CompCode,&Reason);
```

5. Wait until the user has pressed enter and then stop consumption of messages,

```
MQCTL(Hcon,MQOP_STOP,&ctlo,&CompCode,&Reason);
```

6. Finally disconnect from the queue manager,

```
MQDISC(&Hcon,&CompCode,&Reason);
```

The Asynchronous Put sample program

Learn about running the amqsapt sample and the design of 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 Multiplatforms" on page 1020 for the name of this program on different platforms.

Running the amqsapt sample

This program 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_RESPONSE 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.

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 Multiplatforms" on page 1020 for the names of these programs.

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; 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 Browse sample programs for AIX, Linux, and Windows

Consider using this topic when learning about Browse sample programs on AIX, Linux, and Windows.

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.

The Browse sample programs on IBM i
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.

ALW *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 Multiplatforms” on page 1020 for the names of these programs.

This program takes these positional parameters:

1. The name of the source queue (required)
2. The name of the queue manager (required)
3. An optional parameter for properties (optional)

Use the following environment variables to supply credentials that are used to authenticate with the queue manager:

MQSAMP_USER_ID
Set to the user ID to be used for connection authentication, if you want use a user ID and a password to authenticate with the queue manager. The program prompts for the password to accompany the user ID.

Linux V 9.4.0 AIX **MQSAMP_TOKEN**

Set to a non-blank value if you want to supply an authentication token to authenticate with the queue manager. The program prompts for the authentication token. Authentication tokens can be used only by the **amqsbcgc** sample that uses client bindings.

To run these programs, enter one of the following commands:

- `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)

Table 162. Permissible values for the property parameter

Value	Behavior
0	Default behavior. 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 quotation marks, and byte strings are surrounded with X and single quotation marks, as for the message descriptor.

2	MQGMO_NO_PROPERTIES is specified, so that only message descriptor properties will be returned.
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 an IBM MQ 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.

For an example of the output from this utility, see Browsing queues.

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 "Building a procedural application" on page 962 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.

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. Also, learn about running the amqscnxc sample.

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-executable files are supplied.

Running the amqscnxc sample

The command-line syntax of the Connect sample program is:

```
amqscnxc [-x ConnName [-c SvrconnChannelName]] [-u User] [QMgrName]
```

The parameters are optional and their order is not important except for QMgrName, which, if specified, must come last. The parameters are:

ConnName
The TCP/IP connection name of the server queue manager

If you do not specify the TCP/IP connection name, MQCONNX is issued with the *ClientConnPtr* set to NULL.

SvrconnChannelName
The name of the server connection channel

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.

User
The user name to be used for connection authentication

If you specify this the program will prompt for a password to accompany that user ID.

QMgrName
The name of the target queue manager

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 Data-Conversion sample program

The data-conversion sample program is a skeleton of a data conversion exit routine. Learn about the design of the data-conversion sample.

See "Features demonstrated in the sample programs on Multiplatforms" on page 1020 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.

Database coordination samples

Two samples are provided that demonstrate how IBM MQ can coordinate both IBM MQ updates and database updates within the same unit of work.

These samples are:

1. AMQSXAS0 (in C) or AMQ0XAS0 (in COBOL), which updates a single database within an IBM 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 an IBM 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 IBM MQ calls, and where and when to connect to a database.

Figure 128 on page 1043 shows how the samples provided are used to update databases:

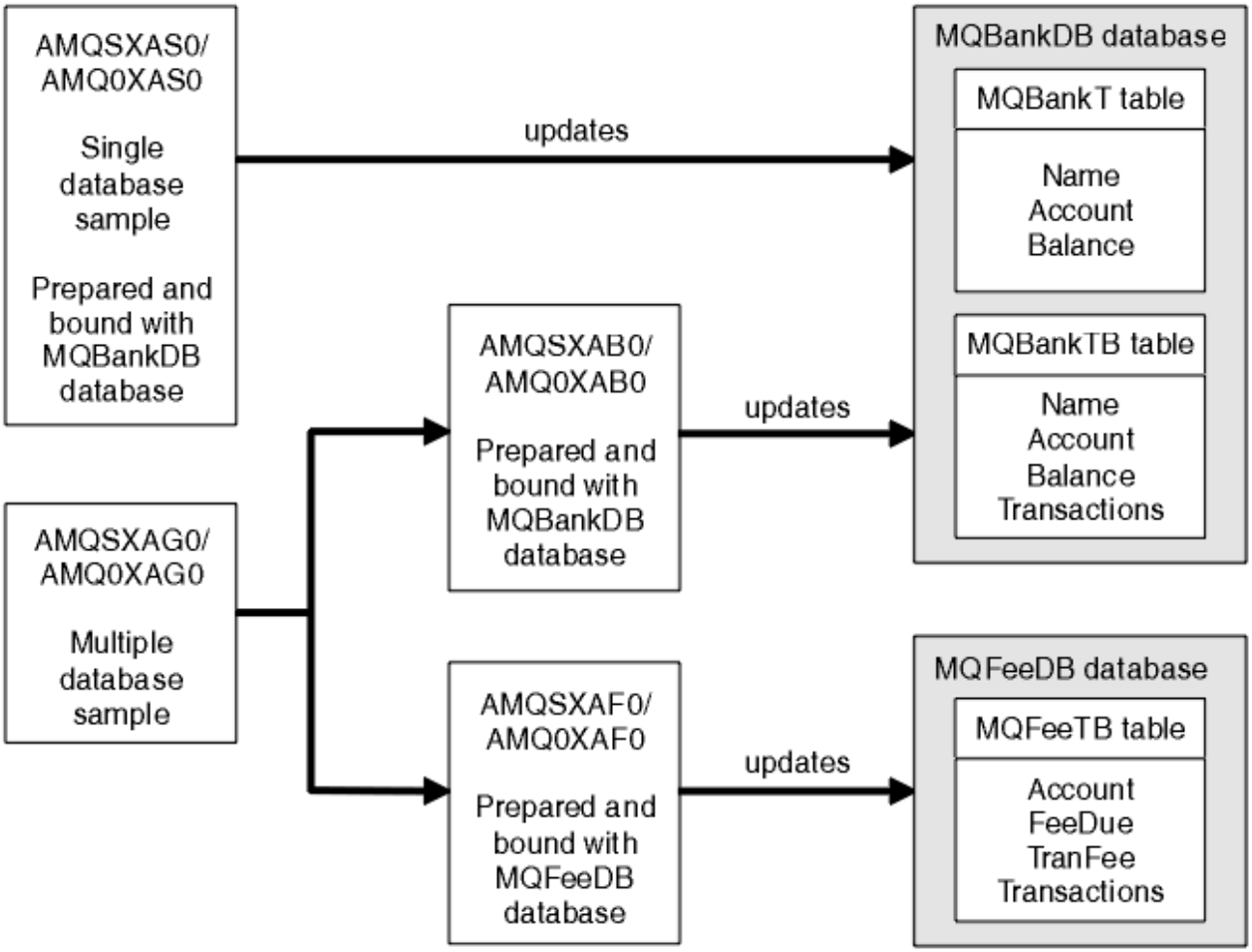


Figure 128. The database coordination samples

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 IBM 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.
```

Enter data into 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
Learn about precompiling, compiling, and linking samples in C and COBOL.

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 typical method for your database product.

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 symbols *DB2TOP* and *MQ_INSTALLATION_PATH*. *DB2TOP* represents the installation directory for the Db2 product. *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.

- AIX On AIX, the directory path is:

```
/usr/lpp/db2_05_00
```

- Windows On Windows systems, the directory path depends on the path chosen when installing the product. If you chose the default settings the path is this:

```
c:\sqllib
```

Note: Before issuing the link command on Windows systems, ensure that the LIB environment variable contains paths to the Db2 and IBM MQ libraries.

Copy the following files into a temporary directory:

- The `amqsxag0.c` file from your IBM MQ installation

  Note: This file can be found in the following directories:

  - Linux AIX On AIX and Linux systems:

    ```
    MQ_INSTALLATION_PATH/samp/xatm
    ```

  - Windows On Windows systems:

    ```
    MQ_INSTALLATION_PATH\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 AIX

```
xlc_r -I MQ_INSTALLATION_PATH/inc -I DB2TOP/include -c -o
FILENAME.o FILENAME.c
```

- Windows Windows systems

```
cl /c /I MQ_INSTALLATION_PATH\tools\c\include /I DB2TOP\include
FILENAME.c
```

Build the `amqsxag0` executable file using the following link command for the platform that you are using:

- AIX AIX

```
xlc_r -H512 -T512 -L DB2TOP/lib -ldb2 -L MQ_INSTALLATION_PATH/lib
-lmqm util.o amqsxaf0.o amqsxab0.o amqsxag0.o -o amqsxag0
```

- Windows Windows systems

```
link util.obj amqsxaf0.obj amqsxab0.obj amqsxag0.obj mqm.lib db2api.lib
/out:amqsxag0.exe
```

Build the `amqsxas0` executable file using the following compiling and linking commands for the platform that you are using:

- AIX

```
xlc_r -H512 -T512 -L DB2TOP/lib -ldb2
-L MQ_INSTALLATION_PATH/lib -lmqm util.o amqsxas0.o -o amqsxas0
```

- Windows systems

```
link util.obj amqsxas0.obj mqm.lib db2api.lib /out:amqsxas0.exe
```

Additional information

AIX If you are working on AIX and want to access Oracle, use the xlc_r compiler and link to libmqm_r.a.

Running the samples
Use this information to learn how to configure the queue manager before running database coordination samples on C and COBOL.

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 Scenario 1: Queue manager performs the coordination.

The following titles provide information about how to run samples in C and COBOL:

- “C samples” on page 1047
- “COBOL samples” on page 1048

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.

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 IBM MQ error log file.

For more information on how the dead-letter handler works, and how you run it, see Processing messages on an IBM MQ dead-letter queue or the *System Management Guide* for your platform.

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 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 Multiplatforms" on page 1020 for the names of these programs.

The programs are intended to run as triggered programs.

On IBM i, AIX, Linux, 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.

IBM i On IBM i, 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; for details, see Change MQ Process (CHGMQMPRC).) 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, this is referred to as 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.

IBM i On IBM i, the program can also respond to messages sent to the queue from platforms other than IBM MQ for IBM i, although no sample is supplied for this situation. To make the ECHO program work:

- 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 IBM MQ for IBM i sending channel, if the program that you have written does not provide similar conversion for the reply.

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 Multiplatforms" on page 1020 for the names of these programs.

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.

Running the amqsget and amqsgetc samples

These programs each take the following positional 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 the *MQSERVER* environment variable or the client channel definition file.
3. The open options (optional)

   If open options are not specified, the sample uses a value of 8193 which is the combination of these two options:

   - MQOO_INPUT_AS_Q_DEF
   - MQOO_FAIL_IF_QUIESCING
4. The close options (optional)

   If close options are not specified, the sample uses a value of 0 which is MQCO_NONE.

Use the following environment variables to supply credentials that are used to authenticate with the queue manager:

MQSAMP_USER_ID
Set to the user ID to be used for connection authentication, if you want use a user ID and a password to authenticate with the queue manager. The program prompts for the password to accompany the user ID.

MQSAMP_TOKEN
Set to a non-blank value if you want to supply an authentication token to authenticate with the queue manager. The program prompts for the authentication token. Authentication tokens can be used only by the **amqsgetc** sample that uses client bindings.

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*.

Using amqsget and amqsgetc

Note that **amqsget** performs a local connection to the queue manager, using shared memory to attach to the queue manager, and as such can only be run on the system the queue manager resides, whereas **amqsgetc** performs a client style connection (even if connecting to a queue manager on the same system).

When using **amqsgetc** you need to provide the application details of how to actually reach the queue manager, in terms of the queue manager host or IP address and queue manager listener port.

Normally this is done either using the **MQSERVER** environment variable or by defining connection details using a client channel definition table,which can also be provided to **amqsgetc** using environment variables; for example, see MQCCDTURL.

An example using **MQSERVER**, connecting to a queue manager locally, which has a listener running on port 1414 and using the default server connection channel is:

```
export MQSERVER="SYSTEM.DEF.SVRCONN/TCP/ localhost(1414)"
```

High availability sample programs

The **amqsghac**, **amqsphac**, and **amqsmhac** high availability sample programs use automated client reconnection to demonstrate recovery following the failure of a queue manager. **amqsfhac** checks that a queue manager using networked storage maintains data integrity following a failure.

The **amqsghac**, **amqsphac**, and **amqsmhac** programs are started from the command line, and can be used in combination to demonstrate reconnection after the failure of one instance of a multi-instance queue manager.

Alternatively, you can also use the **amqsghac**, **amqsphac**, and **amqsmhac** samples to demonstrate client reconnection to single instance queue managers, typically configured into a queue manager group.

To keep the example simple, so it is easy to configure, you are shown the sample programs reconnecting to a single instance queue manager that is started, stopped and then restarted again; see “Set up and control the queue manager” on page 1054.

Use **amqsfhac** in parallel with **amqmfsck** to check file system integrity. See **amqmfsck** (file system check) and Verifying shared file system behavior for more information.

amqsphac *queueName* [*qMgrName*]

- **amqsphac** is an IBM MQ MQI client application. It puts a sequence of messages to a queue with a two second delay between each message and displays events sent to its event handler.
- No sync point is used to put messages to the queue.

- Reconnection can be made to any queue manager in the same queue manager group.

amqsghac *queueName* [*qMgrName*]

- **amqsghac** is an IBM MQ MQI client application. It gets messages from a queue and displays events sent to its event handler.
- No sync point is used to get messages from the queue.
- Reconnection can be made to any queue manager in the same queue manager group.

amqsmhac -s *sourceQueueName* -t *targetQueueName* [-m *qMgrName*] [-w *waitInterval*]

- **amqsmhac** is an IBM MQ MQI client application. It copies messages from one queue to another with a default wait interval of 15 minutes after the last message that is received before the program finishes.
- The messages are copied within sync point.
- Reconnection can be made only to the same queue manager.

amqsfhac *QueueManagerName QueueName SideQueueName InTransactionCount RepeatCount* (0 | 1 | 2)

- **amqsfhac** is an IBM MQ MQI client application. It checks that an IBM MQ multi-instance queue manager using networked storage, such as a NAS or a cluster file system, maintains data integrity. Follow the steps to run **amqsfhac** in Verifying shared file system behavior.
- It uses the `MQCNO_RECONNECT_Q_MGR` option when connecting to *QueueManagerName*. It automatically reconnects when the queue manager fails over.
- It puts *InTransactionCount*RepeatCount* persistent messages to *QueueName* during which time you cause the queue manager to fail over any number of times. **amqsfhac** reconnects to the queue manager each time, and continues. The test is to make sure that no messages are lost.
- *InTransactionCount* messages are put within each transaction. The transaction is repeated *RepeatCount* number of times. If a failure occurs within a transaction, **amqsfhac** rolls back and resubmits the transaction when **amqsfhac** reconnects to the queue manager.
- It also puts messages to *SideQueueName*. It uses *SideQueueName* to check whether the all the messages are committed or rolled back from *QueueName* successfully. If it detects an inconsistency, it writes out an error message.
- Vary the amount of output tracing from **amqsfhac** by setting the last parameter to `(0|1|2)`.

  0
  Least output.

  1
  Middling output.

  2
  Most output.

Configuring a client connection

You need to configure a client and server connection channel to run the samples. The client verification procedure explains how to set up a client test environment.

Alternatively, use the configuration provided in the following example.

Example using amqsghac, amqsphac, and amqsmhac

The example demonstrates reconnectable clients using a single instance queue manager.

Messages are placed on the queue SOURCE by **amqsphac**, transferred to TARGET by **amqsmhac**, and retrieved from TARGET by **amqsghac** ; see Figure 129 on page 1054.

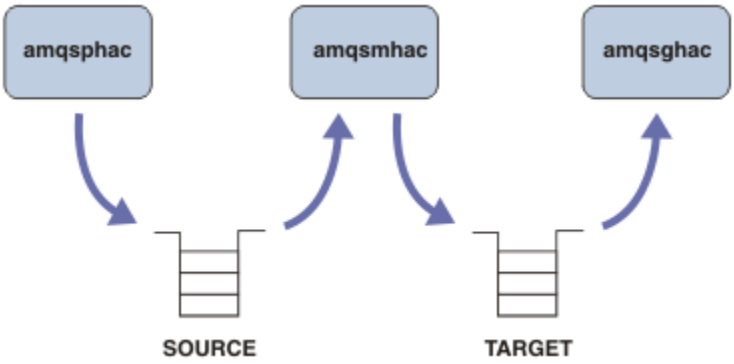


Figure 129. Reconnectable client samples

Follow these steps to run the samples.

1. Create a file `hasamples.tst` containing the commands:

```
DEFINE QLOCAL(SOURCE) REPLACE
DEFINE QLOCAL(TARGET) REPLACE
DEFINE CHANNEL(CHANNEL1) CHLTYPE(SVRCONN)  TRPTYPE(TCP) +
       MCAUSER(MUSR_MQADMIN) REPLACE
DEFINE CHANNEL(CHANNEL1) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
       CONNAME('LOCALHOST(2345)') QMNAME(QM1) REPLACE
ALTER  LISTENER(SYSTEM.DEFAULT.LISTENER.TCP) TRPTYPE(TCP) +
       PORT(2345)
START  LISTENER(SYSTEM.DEFAULT.LISTENER.TCP)
START  CHANNEL(CHANNEL1)
```

2. Type the following commands at a command prompt:
   a. `crtmqm QM1`
   b. `strmqm QM1`
   c. `runmqsc QM1 < hasamples.tst`
3. Set the environment variable **MQCHLLIB** to the path to the `AMQCLCHL.TAB` client channel definition file; for example, `SET MQCHLLIB=C:\IBM\MQ\MQ7\Data\qmgrs\QM1\@ipcc`.
4. Open three new windows with **MQCHLLIB** set; for example on Windows, type **start** three times at the previous command prompt starting each program in one of the windows. See step “5” on page 1055 in “Set up and control the queue manager” on page 1054.)
5. Type the command `endmqm -r -p QM1` to stop the queue manager, and then allow the clients to reconnect.
6. Type the command `strmqm QM1` to restart the queue manager.

The results from running the **amqsghac**, **amqsphac**, and **amqsmhac** samples on Windows are shown in the following examples.

Set up and control the queue manager

1. Create the queue manager.

```
C:\> crtmqm QM1
IBM MQ queue manager created.
Directory 'C:\IBM\MQ\MQ7\Data\qmgrs\QM1' created.
Creating or replacing default objects for QM1.
Default objects statistics : 67 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.
```

   Remember the data directory to set the **MQCHLLIB** variable later.

2. Start the queue manager.

```
C:\> strmqm QM1

IBM MQ queue manager 'QM1' starting.
5 log records accessed on queue manager 'QM1' during the log replay phase.
Log replay for queue manager 'QM1' complete.
Transaction manager state recovered for queue manager 'QM1'.
IBM MQ queue manager 'QM1' started.
```

3. Create the queues and channels, modify the listener port, and start the listener and channel.

```
C:\> runmqsc QM1 < hasamples.tst

5724-H72 (C) Copyright IBM Corp. 1994, 2024.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.

     1 : DEFINE QLOCAL(SOURCE) REPLACE
AMQ8006: IBM MQ queue created.
     2 : DEFINE QLOCAL(TARGET) REPLACE
AMQ8006: IBM MQ queue created.
     3 : DEFINE CHANNEL(CHANNEL1) CHLTYPE(SVRCONN)   TRPTYPE(TCP) MCAUSER(MUSR_MQADMIN)
REPLACE
AMQ8014: IBM MQ channel created.
     4 : DEFINE CHANNEL(CHANNEL1) CHLTYPE(CLNTCONN)  TRPTYPE(TCP) CONNAME('LOCALHOST(2345)')
QMNAME(QM1) REPLACE
AMQ8014: IBM MQ channel created.
     5 : ALTER LISTENER(SYSTEM.DEFAULT.LISTENER.TCP) TRPTYPE(TCP) PORT(2345)
AMQ8623: IBM MQ listener changed.
     6 : START LISTENER(SYSTEM.DEFAULT.LISTENER.TCP)
AMQ8021: Request to start IBM MQ Listener accepted.
     7 : START CHANNEL(CHANNEL1)
AMQ8018: Start IBM MQ channel accepted.
7 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

4. Make the client channel table known to the clients.

   Use the data directory returned from the **crtmqm** command in step “1” on page 1054, and add the directory `@ipcc` to it to set the **MQCHLLIB** variable.

```
C:\> SET MQCHLLIB=C:\IBM\MQ\MQ7\Data\qmgrs\QM1\@ipcc
```

5. Start the sample programs in the other windows

```
C:\> start amqsphac SOURCE QM1
C:\> start amqsmhac -s SOURCE -t TARGET -m QM1
C:\> start amqsghac TARGET QM1
```

6. End the queue manager and restart it again.

```
C:\> endmqm -r -p QM1

Waiting for queue manager 'QM1' to end.
IBM MQ queue manager 'QM1' ending.
IBM MQ queue manager 'QM1' ended.


C:\> strmqm QM1

IBM MQ queue manager 'QM1' starting.
5 log records accessed on queue manager 'QM1' during the log replay phase.
Log replay for queue manager 'QM1' complete.
Transaction manager state recovered for queue manager 'QM1'.
IBM MQ queue manager 'QM1' started.
```

amqsphac

```
Sample AMQSPHAC start
target queue is SOURCE
message Message 1
message Message 2
16:25:22 : EVENT : Connection Reconnecting (Delay: 0ms)
16:25:45 : EVENT : Connection Reconnecting (Delay: 0ms)
16:26:02 : EVENT : Connection Reconnectedmessage
Message 3
message Message 4
message Message 5
```

amqsmhac

```
Sample AMQSMHA0 start
16:25:22 : EVENT : Connection Reconnecting (Delay: 0ms)
16:25:45 : EVENT : Connection Reconnecting (Delay: 0ms)
16:26:02 : EVENT : Connection Reconnected
No more messages.
Sample AMQSMHA0 end
C:\>
```

amqsghac

```
Sample AMQSGHAC start
message Message 1
message Message 2
16:25:22 : EVENT : Connection Reconnecting (Delay: 0ms)
16:25:45 : EVENT : Connection Reconnecting (Delay: 0ms)
16:26:02 : EVENT : Connection Reconnected
message Message 3
message Message 4
message Message 5
```

Related tasks

Verifying shared file system behavior

Related reference

amqmfsck (file system check)

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 Multiplatforms" on page 1020 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 IBM MQ for Multiplatforms. This structure contains the name of a target queue with attributes that are to be inquired upon. 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. IBM i For IBM i, you can use the CHGMQMPRC command for this; for details, see Change MQ Process (CHGMQMPRC). 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:

- ALW For AIX, Linux, and Windows, start the **runmqtrm** program in one session, then start the `amqsreq` program in another.
- IBM i For IBM i, 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.

IBM i On IBM i, 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 Inquire Properties of a Message Handle sample program

AMQSIQMA is a sample C program to inquire properties of a message handle from a message queue, and is an example of the use of the MQINQMP API call.

This sample creates a message handle and puts it into the MsgHandle field of the MQGMO structure. The sample then gets one message and inquires and prints all properties with which the message handle was populated.

```
C:\Program Files\IBM\MQ\tools\c\Samples\Bin >amqsiqm Q QM1
Sample AMQSIQMA start
property name MyProp value MyValue
message text Hello world!
Sample AMQSIQMA end
```

The Publish/Subscribe sample programs

The publish/subscribe sample programs demonstrate the use of the publish and subscribe features in IBM MQ.

There are three C language sample programs illustrating how to program to the IBM MQ publish/subscribe interface. There are some C samples that use older interfaces, and there are Java samples.

The Java samples use the IBM MQ publish/subscribe interface in com.ibm.mq.jar and the JMS publish/subscribe interface in com.ibm.mqjms. The JMS samples are not covered in this topic.

C

Find the publisher sample `amqspub` in the C samples folder. Run it with any topic name you like as the first parameter, followed by an optional queue manager name. For example, `amqspub mytopic QM3` . There is also a client version called `amqspubc`. If you choose to run the client version, first see "Configuring a queue manager to accept client connections on Multiplatforms" on page 1028 for details.

The publisher connects to the default queue manager and responds with the output, `target topic is mytopic` . Every line you enter into this window from now onwards is published to `mytopic` .

Open another command window in the same directory, and run the subscriber program, `amqssub`, supplying it with the same topic name, and an optional queue manager name. For example, `amqssub mytopic QM3` .

The subscriber responds with the output, `Calling MQGET : 30 seconds wait time` . From now onwards, lines you type into the publisher appear in the output of the subscriber.

Start another subscriber in another command window, and watch both subscribers receive publications.

For full documentation of the parameters, including setting options, refer to the sample source code. The values for the subscriber options field is described in the following topic: Options (MQLONG).

There is another subscriber sample `amqssbx`, which offers additional subscription options as command line switches.

Type `amqssbx -d mysub -t mytopic -k` to invoke the subscriber using durable subscriptions that are retained after the subscriber has terminated.

Test the subscription by publishing another item using the publisher. Wait for 30 seconds for the subscriber to terminate. Publish some more items under the same topic. Restart the subscriber. The last item published while the subscriber was not running is displayed by the subscriber immediately it is restarted.

C legacy

There is an additional set of C samples which demonstrate queued commands. Some of these samples were originally shipped as part of the MQ0C Supportpac. The capabilities the samples demonstrate are fully supported, for compatibility reasons.

We discourage you from using the queued command interface. It is much more complex than the publish/subscribe API, and there is no compelling functional reason to program complex queued commands. However, you might find the queued approach more suitable, perhaps because you are already using the interface, or because your programming environment makes it easier to build a complex message and call a generic MQPUT, rather than constructing different calls to MQSUB.

The additional samples are located in the `pubsub` subdirectory in the `samples` folder.

There are six types of sample listed in Table 163 on page 1058.

Table 163. Categories of legacy publish/subscribe sample C programs

Category	Programs	Comments
RFH1	`amqssr1a.c` `amqspr1a.c`	Simple publish/subscribe example built using RFH1 format messages.
RFH2	`amqssr2a.c` `amqspr2a.c`	Simple publish/subscribe example built using RFH2 format messages.

Table 163. Categories of legacy publish/subscribe sample C programs (continued)		
Category	**Programs**	**Comments**
MQAI samples	`amqsppca.c` `amqsspca.c`	Simple publish/subscribe example built using PCF commands and the MQAI command interface.
MA0C Results service using RFH1	`amqsgama.c` `amqsresa.c`	Results service built using RFH1 headers 1. Requires the queues defined in `amqsgama.tst` and `amqsresa.tst` 2. `amqsresa` must be started before `amqsgama`
MA0C Results service using RFH2	`amqsgr2a.c` `amqsrr2a.c`	Results service built using RFH2 headers 1. Requires the queues defined in `amqsgama.tst` and `amqsresa.tst` 2. `amqsresa` must be started before `amqsgama`
Routing exit publish/ subscribe sample	`amqspsra.c`	Demonstrates how to change the queue or queue manager destination for a publish/subscribe message in a routing exit.

Sample program for Java

The Java sample `MQPubSubApiSample.java` combines publisher and subscribers in a single program. Its source and compiled class files are found in the `wmqjava` samples folder.

If you choose to run in client mode, first see “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028 for details.

Run the sample from the command line using the Java command, if you have a Java environment configured. You can also run the sample from the IBM MQ Explorer Eclipse workspace that has a Java programming workbench already set up.

You might need to change some of the sample program's properties to run it. You do this by providing parameters to the JVM, or editing the source.

The instructions in “Running the MQPubSubApiSample Java sample” on page 1059 show how to run the sample from the Eclipse workspace.

Running the MQPubSubApiSample Java sample
How to run the MQPubSubApiSample using the Java Development Tools from the Eclipse platform.

Before you begin

Open the Eclipse workbench. Create a new workspace directory and select it. Close the welcome window.

Follow the steps in “Configuring a queue manager to accept client connections on Multiplatforms” on page 1028 prior to running as a client.

About this task

The Java publish/subscribe sample program is an IBM MQ MQI client Java program. The sample runs without modification using a default queue manager listening on port 1414. The task describes this simple case, and indicates in general terms how to provide parameters and modify the sample to suit different IBM MQ configurations. The example is illustrated running on Windows. The file paths will differ on other platforms.

Procedure

1. Import the Java sample programs
   a) In the workbench, click **Window** > **Open perspective** > **Other** > **Java** and click **OK**.
   b) Switch to the **Package Explorer** view.
   c) Right-click in the white-space in the **Package Explorer** view. Click **New** > **Java project**.
   d) In the **Project name** field type `MQ Java Samples`. Click **Next**.
   e) In the **Java Settings** panel, switch to the **Libraries** tab.
   f) Click **Add External JARs**.
   g) Browse to *MQ_INSTALLATION_PATH* `\java\lib` where *MQ_INSTALLATION_PATH* is the IBM MQ installation folder and select `com.ibm.mq.jar` and `com.ibm.mq.jmqi.jar`
   h) Click **Open** > **Finish.**
   i) Right-click `src` in the **Package Explorer** view.
   j) Select **Import...** > **General** > **File System** > **Next** > **Browse**... and browse to the path *MQ_INSTALLATION_PATH* `\tools\wmqjava\samples` where *MQ_INSTALLATION_PATH* is the IBM MQ installation directory.
   k) On the **Import** panel, Figure 130 on page 1061, click `samples` (do not select the check box).
   l) Select `MQPubSubApiSample.java`. The **Into folder** field should contain `MQ Java Samples/src`. Click **Finish**.

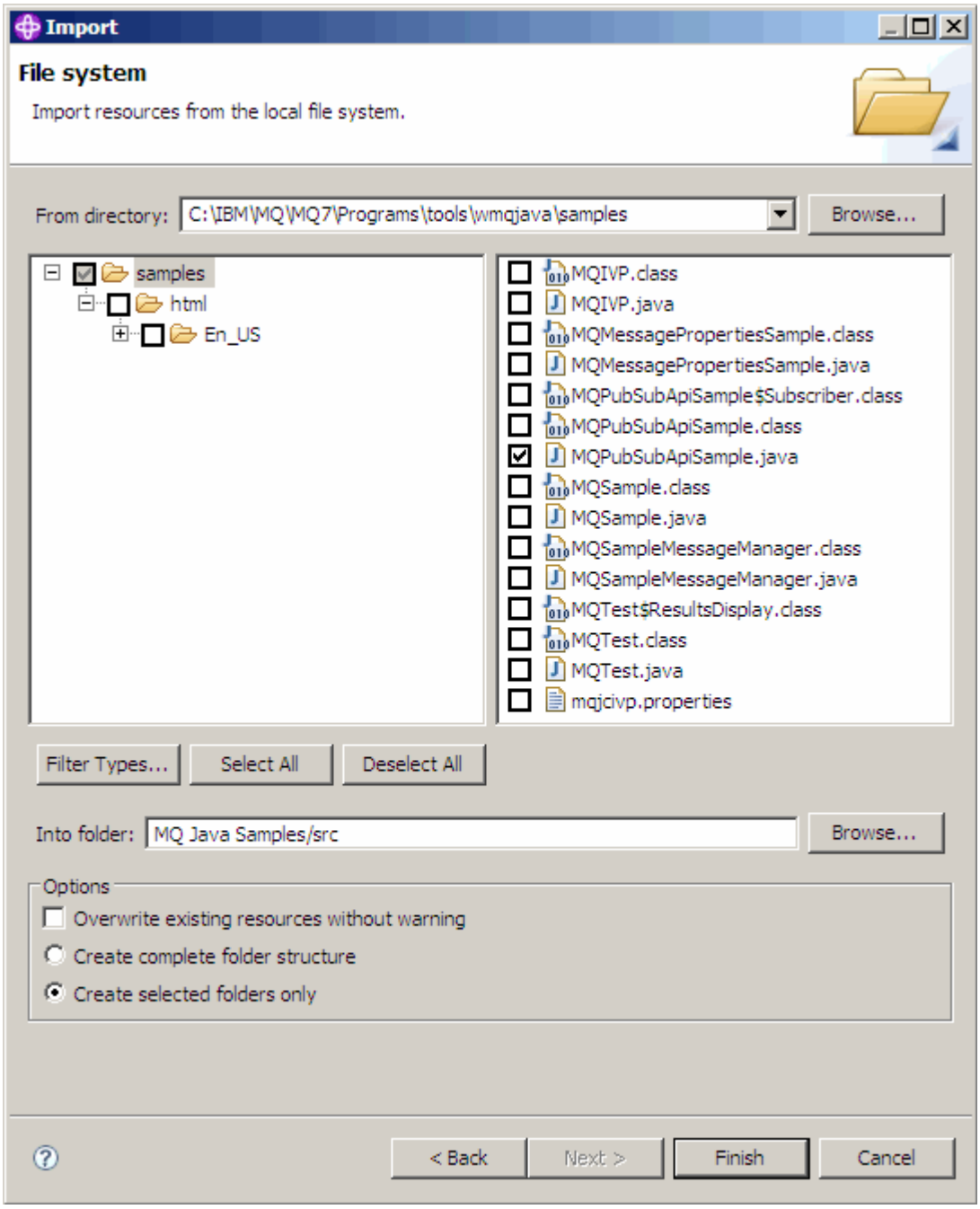


Figure 130. File system import

2. Run the publish/subscribe sample program.

   There are two ways to run the program, depending on whether you need to change the default parameters.

   - The first choice runs the program without making any changes:
     - In the workspace main menu, expand the `src` folder. Right-click **MQPubSubApiSample.java** **Run-as** > **1. Java Application**
   - The second choice runs the program with parameters or with modified source code for your environment:
     - Open `MQPubSubApiSample.java` and study the `MQPubSubApiSample` constructor.

– Modify the attributes of the program.

  These attributes are modifiable using the -D JVM switch, or by providing a default value for the System property by editing the source code.

  - topicObject
  - queueManagerName
  - subscriberCount

  These attributes are changeable only by editing the source code in the constructor.

  - hostname
  - port
  - channel

  To set System properties, code a default value in the accessor, for example:

```
queueManagerName = System.getProperty("com.ibm.mq.pubSubSample.queueManagerName",
"QM3");
```

  Or provide the parameter to the JVM using the `-D` option, as shown in the following steps:

a. Copy the full name of the System.Property you want to set, for example: `com.ibm.mq.pubSubSample.queueManagerName`.

b. In the workspace, right-click **Run** > **Open Run Dialog**. Double-click Java Application in **Create, Manage and Run applications** and click the **(x) = Arguments** tab.

c. In the **VM arguments:** pane, type `-D` and paste the System.property name, `com.ibm.mq.pubSubSample.queueManagerName`, followed by `=QM3`. Click **Apply** > **Run**.

d. Add further arguments as a comma separated list, or as additional lines in the pane, without comma separators.

   For example: `-Dcom.ibm.mq.pubSubSample.queueManagerName=QM3, -Dcom.ibm.mq.pubSubSample.subscriberCount=6`.

The Publish Exit sample program

AMQSPSE0 is a sample C program of an exit to intercept a publication before it is delivered to a subscriber. The exit can then, for example, alter the message headers, payload or destination, or prevent the message being published to a subscriber.

To run the sample, perform the following tasks:

1. Configure the queue manager:
   - Linux AIX On AIX and Linux systems add a stanza like this to the `qm.ini` file:

     ```
     PublishSubscribe:
     PublishExitPath=Module
     PublishExitFunction=EntryPoint
     ```

     where the module is *MQ_INSTALLATION_PATH*`/samp/bin/amqspse`. *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.
   - Windows On Windows set the equivalent attributes in the registry.
2. Make sure the Module is accessible to IBM MQ.
3. Restart the queue manager to pick up the configuration.
4. In the application process to be traced, describe where the trace files should be written to. For example:

- Linux AIX On AIX and Linux systems, ensure that the directory `/var/mqm/trace` exists and export the **MQPSE_TRACE_LOGFILE** environment variable:

```
export MQPSE_TRACE_LOGFILE=/var/mqm/trace/PubTrace
```

- Windows On Windows, ensure that the directory `C:\temp` exists and set the **MQPSE_TRACE_LOGFILE** environment variable:

```
set MQPSE_TRACE_LOGFILE=C:\temp\PubTrace
```

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 Multiplatforms" on page 1020 for the names of these programs.

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.

Running the Put sample programs

Running the amqsput and amqsputc samples

ALW

The **amqqsput** sample is the program for putting messages using local bindings, and the **amqsputc** sample is the program for putting messages using client bindings. These programs each take the following positional 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 the *MQSERVER* environment variable or the client channel definition file.
3. The open options (optional)

   If open options are not specified, the sample uses a value of 8208 which is the combination of these two options:

   - MQOO_OUTPUT
   - MQOO_FAIL_IF_QUIESCING
4. The close options (optional)

   If close options are not specified, the sample uses a value of 0 which is MQCO_NONE.
5. The name of the target queue manager (optional)

   If a target queue manager is not specified, the `ObjectQMgrName` field in the MQOD will be left blank.

6. The name of the dynamic queue (optional)

   If a dynamic queue name is not specified, the `DynamicQName` field in the MQOD will be left blank.

Use the following environment variables to supply credentials that are used to authenticate with the queue manager:

MQSAMP_USER_ID
Set to the user ID to be used for connection authentication, if you want use a user ID and a password to authenticate with the queue manager. The program prompts for the password to accompany the user ID.

Linux V 9.4.0 AIX **MQSAMP_TOKEN**
Set to a non-blank value if you want to supply an authentication token to authenticate with the queue manager. The program prompts for the authentication token. Authentication tokens can be used only by the **amqsputc** sample that uses client bindings.

To run these programs, enter one of the following commands:

- `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

ALW

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 (IBM i)

IBM i

The C program AMQSPUT4, available only for the IBM i platform, 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 (IBM i)

IBM i

The COBOL program AMQ0PUT4, available only on the IBM i platform, 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.

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 IBM 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 the sample, amqsxrm.

The 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 platform except IBM i.
- IBM i If the receiver is running under IBM i, use amqsxrm4.

.

IBM i *Notes for IBM i 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 IBM i 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.

Multi *Running the Reference Message samples*

Use this example to find out how to run Reference Message sample application AMQSPRM on AIX, Linux, and Windows, or AMQSPRMA on IBM i. The example shows how Reference Messages can be put to a queue, received by message exits, and taken from a queue.

The Reference Message samples run as follows:

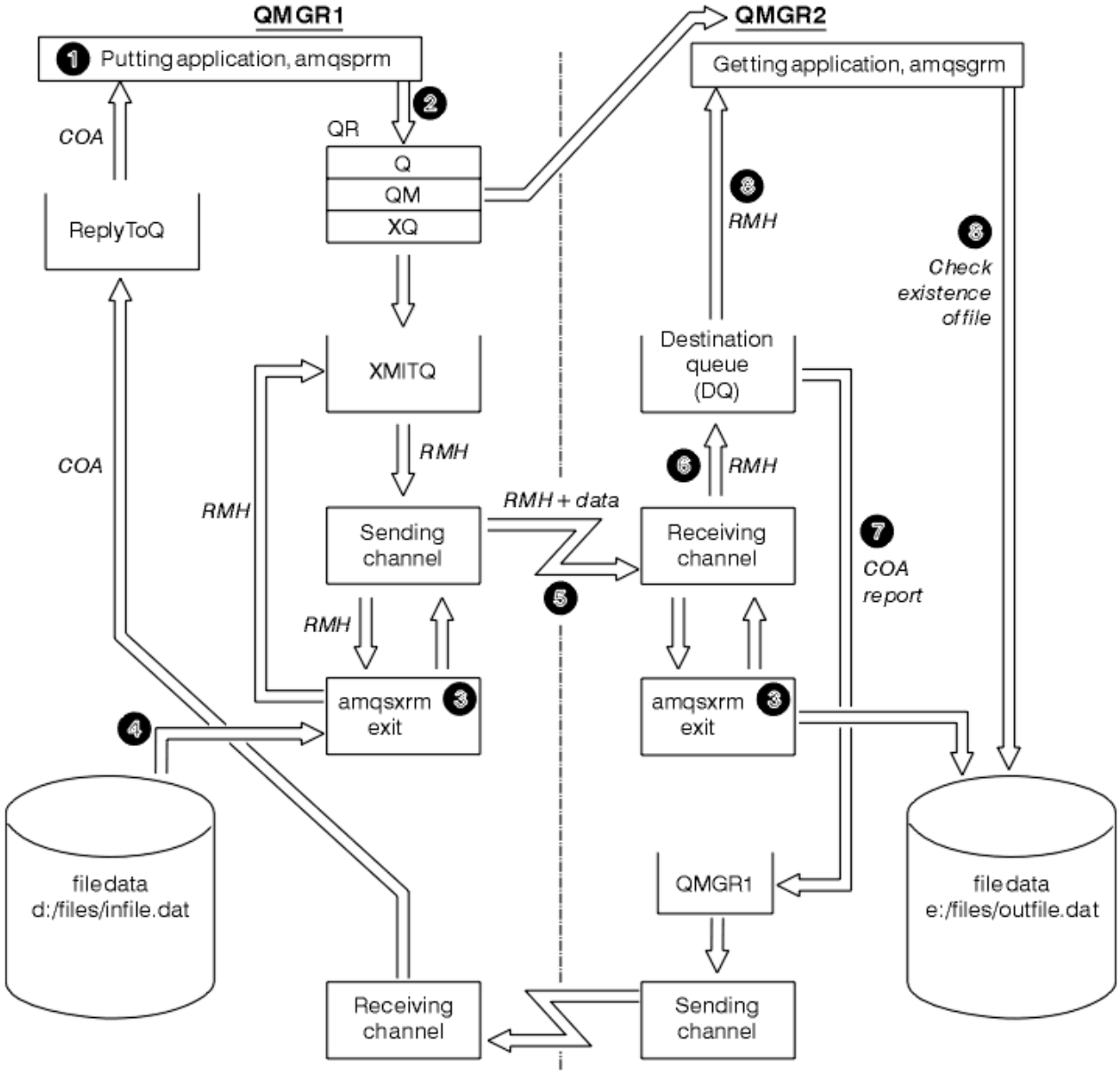


Figure 131. Running the Reference Message samples

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, this example 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 IBM i). 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 IBM i) 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 IBM MQ product is installed.

IBM i The MQSC definitions have used an AIX style for defining the exits, so if you are using MQSC on IBM i, 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
```

IBM i **IBM i 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)
```

IBM i **IBM i command syntax**

Note: IBM i On IBM i, 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 IBM 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 IBM i) 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 the destination file

   -q
   Name of the queue

   -g
   Name of the 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 previously 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
```

   IBM i users: IBM i On IBM i, complete the following steps:

   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 example /TEST/FILENAME creates a file called FILENAME in the directory TEST.

Note:

IBM i On IBM i, 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
```

Linux AIX For AIX and Linux platforms, you must use two backslashes (\\) instead of one to denote the destination file directory. Therefore, the **amqsprm** 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 are `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.

Windows This is defined on Windows as follows:

```
msgexit(' pathname\amqsxrm.dll(MsgExit)')
```

Linux AIX This is defined on AIX and Linux as follows:

```
msgexit(' pathname/amqsxrm(MsgExit)')
```

If you specify a path name, specify the complete name. If you omit the path name, it is assumed that the program is in the path specified in the `qm.ini` file (or, on IBM MQ for Windows, the path specified in the registry).

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.

Linux AIX On AIX and Linux, 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 IBM 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 topic gives a detailed description of a Put Reference Message sample.

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 IBM 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 AIX and Linux platforms, use two backslashes (\\) instead of one to denote the destination file directory. Therefore, the **amqsprm** 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 the Reference Message Exit sample, use the command for the platform on which IBM MQ is installed.

MQ_INSTALLATION_PATH represents the high-level directory in which IBM MQ is installed.

To compile amqsxrma, use the following commands:

On AIX

AIX

```
xlc_r -q64 -e MsgExit -bE:amqsxrm.exp -bM:SRE -o amqsxrm_64_r
-I MQ_INSTALLATION_PATH/inc -L MQ_INSTALLATION_PATH/lib64 -lmqm_r amqsqrma.c
```

On IBM i

IBM i

```
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

Linux

```
$ gcc -m64 -shared -fPIC -o /var/mqm/exits64/amqsxrma amqsqrma.c -I MQ_INSTALLATION_PATH/inc
-L MQ_INSTALLATION_PATH/lib64 -Wl,-rpath= MQ_INSTALLATION_PATH/lib64 -Wl,-rpath=/usr/lib64
-lmqm_r
```

On Windows

Windows IBM MQ now supplies the mqm library with client packages as well as server packages, so the following example uses `mqm.lib` instead of `mqmvx.lib`:

```
cl amqsqrma.c /link /out:amqsxrm.dll /dll mqm.lib mqm.lib /def:amqsxrm.def
```

Related concepts

"Writing channel-exit programs" on page 929
You can use the following information to help you write channel-exit programs.

Design of the Get Reference Message sample (amqsgrma.c, AMQSGRM4)
This topic explains the design of the Get Reference Message sample.

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 Multiplatforms" on page 1020 for the names of the Request sample programs.

Running the Request sample programs

Running the amqsreq0.c, amqsreq, and amqsreqc samples

The C version of the program takes three parameters:

1. The name of the target server queue (necessary)
2. The name of the queue manager (optional)
3. The reply queue (optional)

For example, enter one of the following:

- `amqsreq myqueue qmanagername replyqueue`
- `amqsreqc myqueue qmanagername`
- `amq0req0 myqueue`

where `myqueue` is the name of the target server queue, `qmanagername` is the name of the queue manager that owns `myqueue`, and `replyqueue` is the name of the reply queue.

If you omit the name of the queue manager, it is assumed that the default queue manager owns the queue. If you omit the name of the reply queue, the default reply queue is provided.

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.

ALW *Running the Request sample using triggering on AIX, Linux, and Windows*

On AIX, Linux, and Windows, start the trigger monitor program RUNMQTRM in one session, and then start the amqsreq program in another session.

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 132 on page 1074 demonstrates how to use the Request and Inquire samples together.

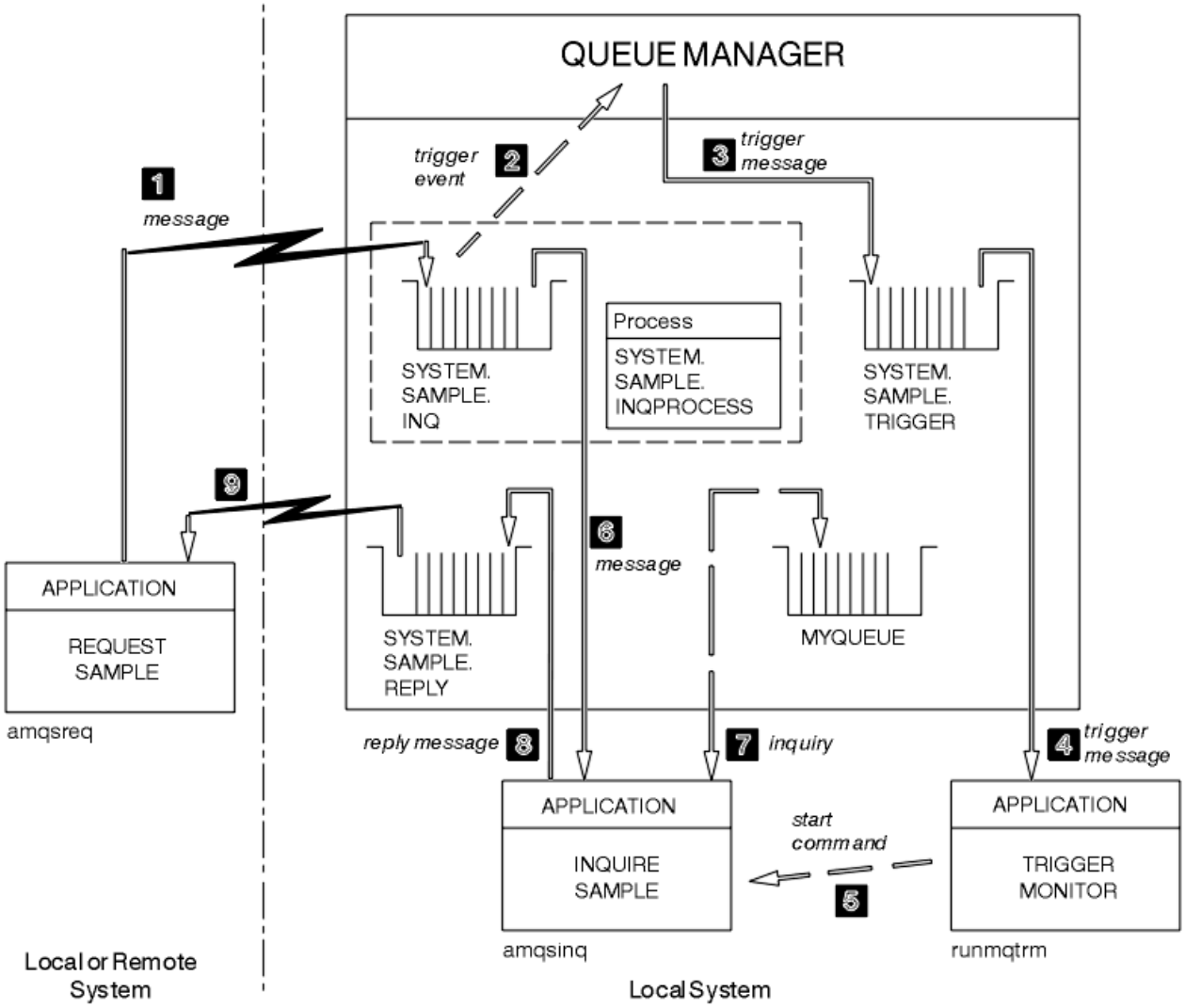


Figure 132. Request and Inquire samples using triggering

In Figure 132 on page 1074 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 132 on page 1074 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 ApplType.

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" on page 1075.

For more information on triggering see "Starting IBM MQ applications using triggers" on page 834.

IBM i *Running the Request sample using triggering on IBM i*

On IBM i, 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 (for details, see Change MQ Process (CHGMQMPRC)) 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 133 on page 1077. 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, for details, see Start MQ Trigger Monitor (STRMQMTRM)) 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 that 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.

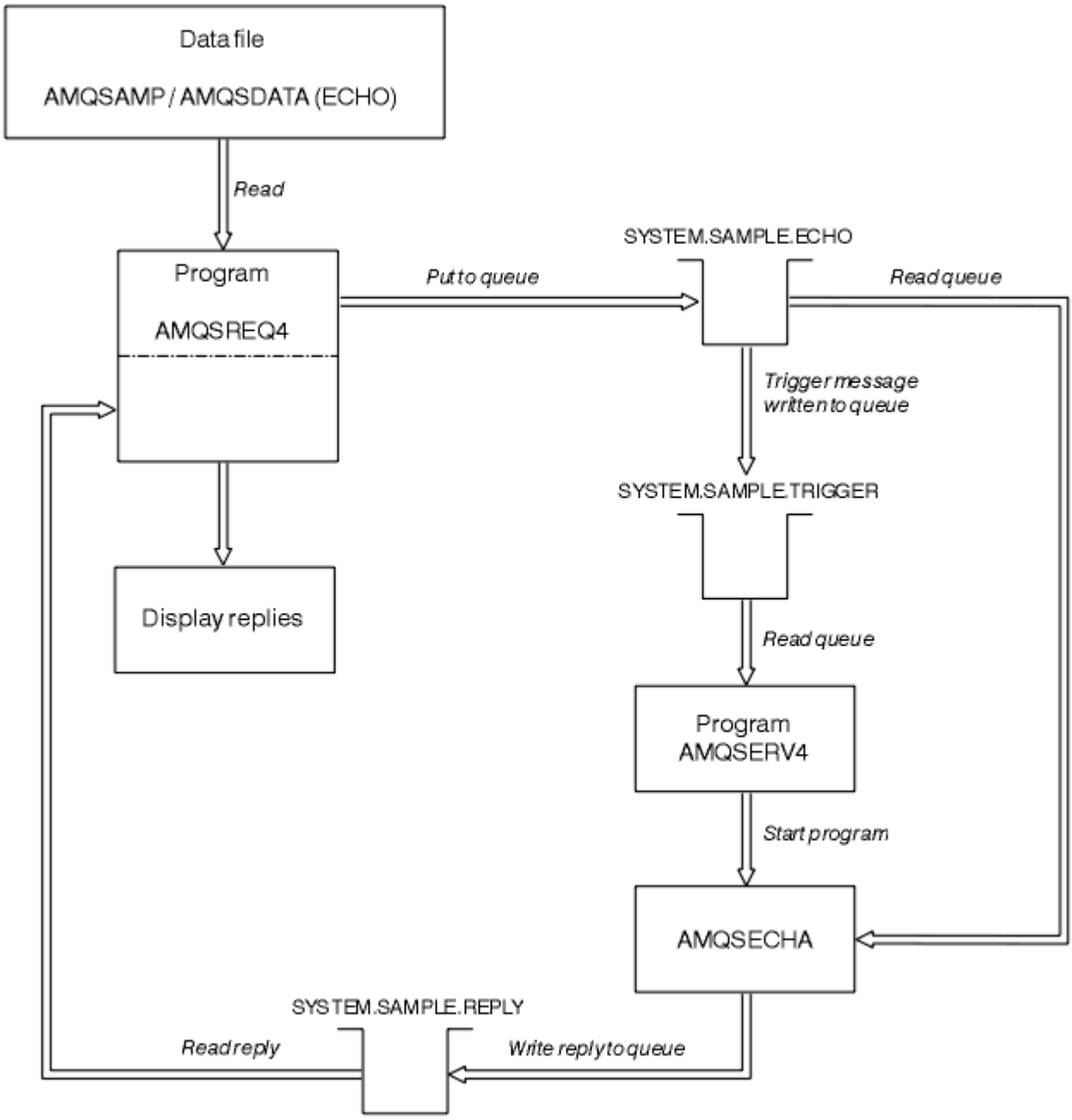


Figure 133. Sample IBM i Client/Server (Echo) program flowchart

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. Also, learn about the design of Set sample programs.

See "Features demonstrated in the sample programs on Multiplatforms" on page 1020 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 with attributes that are to be inquired upon. 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:

- ALW For AIX, Linux, and Windows systems, start the **runmqtrm** program in one session, then start the amqsreq program in another.
- IBM i For IBM i, 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 TLS sample program

AMQSSSLC is a sample C program that demonstrates how to use the MQCNO and MQSCO structures to supply TLS client connection information on the MQCONNX call. This enables a client MQI application to provide the definition of its client connection channel and TLS settings at run time without a client channel definition table (CCDT).

If a connection name is supplied, the program constructs a client connection channel definition in an MQCD structure.

If the stem name of the key repository file is supplied, the program constructs an MQSCO structure; if an OCSP responder URL is also supplied, the program constructs an authentication information record MQAIR structure.

The program then connects to the queue manager using MQCONNX. It inquires and prints out the name of the queue manager to which it connected.

This program is intended to be linked as an MQI client application. However, it can be linked as a regular MQI application, in which case it simply connects to a local queue manager and ignores the client connection information.

If the passphrase to access the key repository is not stashed to a file, you must supply the passphrase to **amqssslc** when the application runs. You can supply the passphrase either by:

- Requesting **amqssslc** to prompt for the passphrase, or
- Using the *MQKEYRPWD* environment variable, or
- Using the **SSLKeyRepositoryPassword** attribute in the client configuration file

For more information about supplying the key repository password to IBM MQ MQI client applications, see Supplying the key repository password for an IBM MQ MQI client on AIX, Linux, and Windows.

amqssslc accepts the following parameters, all of which are optional:

-m QmgrName
Name of the queue manager to connect to

-c ChannelName
Name of the channel to use

-x ConnName
Server connection name

TLS parameters:

-k KeyReposFileName
The name of the key repository file. If the file extension is not supplied, it is assumed to be .kdb. For example:

```
/home/user/client.kdb
C:\User\client.p12
```

-s CipherSpec
The TLS channel CipherSpec string corresponding to the **SSLCIPH** on the SVRCONN channel definition on the queue manager.

-f
Specifies that only FIPS 140-2 certified algorithms must be used.

-b VALUE1[,VALUE2...]
Specifies that only Suite B compliant algorithms must be used. This parameter is a comma-separated list of one or more of the following values: NONE,128_BIT,192_BIT. These values have the same meaning as those for the **MQSUITEB** environment variable, and the equivalent **EncryptionPolicySuiteB** setting in the client configuration file SSL stanza.

-p Policy
Specifies the certificate validation policy to be used. This can be one of the following values:

ANY
Apply each of the certificate validation policies supported by the secure sockets library and accept the certificate chain if any of the policies considers the certificate chain valid. This setting can be used for maximum backwards compatibility with older digital certificates which do not comply with the modern certificate standards.

RFC5280
Apply only the RFC 5280 compliant certificate validation policy. This setting provides stricter validation than the ANY setting, but rejects some older digital certificates.

The default value is ANY.

-l CertLabel
The certificate label to use for the secure connection.

Note: You must specify the value using lowercase characters.

-w
Specifies that **amqssslc** prompts for the key repository passphrase to be supplied.

-i
Specifies that **amqssslc** prompts for the initial key used to encrypt the key repository passphrase to be supplied.

Specify this option if an initial key file was specified when the key repository passphrase was encrypted using the **runmqicred** utility.

OCSP certificate revocation parameter:

-o URL
The OCSP Responder URL

You can also set one of the following environment variables to supply credentials that are used to authenticate with the queue manager:

MQSAMP_USER_ID
Set to the user ID to be used for connection authentication, if you want use a user ID and a password to authenticate with the queue manager. The program prompts for the password to accompany the user ID.

Linux V 9.4.0 AIX **MQSAMP_TOKEN**
Set to a non-blank value if you want to supply an authentication token to authenticate with the queue manager. The program prompts for the authentication token.

Running the TLS sample program
To run the TLS sample program you must first set up your TLS environment. You then run the sample from the command line, supplying a number of parameters.

About this task

The following instructions run the sample program using personal certificates. By varying the command you can, for example, use CA certificates and check their status using an OCSP responder. See the instructions within the sample.

Procedure

1. Create a queue manager with the name QM1. For more information, see crtmqm.
2. Create a key repository for the queue manager. For more information, see Setting up a key repository on AIX, Linux, and Windows.
3. Create a key repository for the client. Call it *clientkey.kdb*.
   Stash the key repository password in a file when you create the key repository.
4. Create a personal certificate for the queue manager. For more information, see Creating a self-signed personal certificate on AIX, Linux, and Windows.
5. Create a personal certificate for the client.
6. Extract the personal certificate from the server key repository and add it to the client repository. For more information, see Extracting the public part of a self-signed certificate from a key repository on AIX, Linux, and Windows, and Adding a CA certificate (or the public part of a self-signed certificate) into a key repository, on AIX, Linux, and Windows systems.
7. Extract the personal certificate from the client key repository and add it to the server key repository.
8. Create a server connection channel using the MQSC command:

   ```
   DEFINE CHANNEL(QM1SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
   SSLCIPH(TLS_RSA_WITH_AES_128_CBC_SHA256)
   ```

   For more information see Server-connection channel
9. Define and start a channel listener on the queue manager. For more information see DEFINE LISTENER and START LISTENER.
10. Run the sample program using the following command:

```
AMQSSSLC -m QM1 -c QM1SVRCONN -x localhost
-k "C:\Program Files\IBM\MQ\clientkey.kdb" -s TLS_RSA_WITH_AES_128_CBC_SHA256
-o http://dummy.OCSP.responder
```

Results

The sample program performs the following actions:

1. Connects to any specified queue manager, or to the default queue manager, using any options specified.
2. Opens the queue manager and inquires on its name.
3. Closes the queue manager.
4. Disconnects from the queue manager.

If the sample program runs successfully, it displays output similar to the following example:

```
Sample AMQSSSLC start
Connecting to queue manager QM1
Using the server connection channel QM1SVRCONN
on connection name localhost.
Using TLS CipherSpec TLS_RSA_WITH_AES_128_CBC_SHA256
Using TLS key repository stem C:\Program Files\IBM\MQ\clientkey
Using OCSP responder URL http://dummy.OCSP.responder
Connection established to queue manager QM1

Sample AMQSSSLC end
```

If the sample program encounters a problem, it displays an appropriate error message, for example if you specify an invalid OCSP responder URL, you receive the following message:

```
MQCONNX ended with reason code 2553
```

For a list of reason codes see API completion and reason codes.

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 Multiplatforms” on page 1020 for the names of these programs.

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 IBM i, AIX, Linux, 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.

- ALW In AIX, Linux, and Windows systems, the *EnvData* field is used as an extension to the invoking command string.
- IBM i In IBM i, it is used as job submission parameters, for example, the job priority or the job description.

Finally, the program closes the initiation queue.

Ending the triggering sample programs on IBM i

IBM i

A trigger monitor program can be ended by the sysrequest option 2 (ENDRQS) or by inhibiting gets from the trigger queue.

If the sample trigger queue is used, the command is:

```
CHGMQMQ QNAME('SYSTEM.SAMPLE.TRIGGER') MQMNAME GETENBL(*NO)
```

Important: Before starting triggering again on this queue, you must enter the following command:

```
CHGMQMQ QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*YES)
```

Running the Triggering sample programs
This topic contains information about running Triggering sample 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

IBM i

This is a trigger monitor for the IBM i environment. It submits one IBM i job for each application to be started. This means that there is additional processing 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; for details, see Start MQ Trigger Monitor (STRMQMTRM).

Running the AMQSERV4 sample

IBM i

This is a trigger server for the IBM i environment. For each trigger message, this server runs the start command in its own job to start the specified application. The trigger server can call CICS transactions.

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, with a few exceptions

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.
- IBM i Calls IBM i applications in its own job (or uses STRCICSUSR to start CICS applications) rather than submitting an IBM i 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.

ALW *Using the TUXEDO samples on AIX, Linux, and Windows*

Learn about the Put and Get sample programs for TUXEDO, and building the server environment in TUXEDO.

Before you begin

Before running these samples, you must build the server environment.

About this task

Note: Throughout this section, the backslash (\) character is used to split long commands over more than one line. Do not enter this character. Enter each command as a single line.

ALW *Building the server environment*
Information about building the server environment for IBM MQ for different platforms.

Before you begin

It is assumed that you have a working TUXEDO environment.

AIX *Building the server environment for AIX (32-bit)*
How to build the server environment for IBM MQ for AIX (32-bit).

Procedure

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, and *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed:

```
$ export CFLAGS="-I MQ_INSTALLATION_PATH/inc -I /APPDIR -L MQ_INSTALLATION_PATH/lib"
$ export LDOPTS="-lmqm"
$ export FIELDTBLS= MQ_INSTALLATION_PATH/samp/amqstxvx.flds
$ export VIEWFILES=/APPDIR/amqstxvx.V
$ export LIBPATH=$TUXDIR/lib: MQ_INSTALLATION_PATH/lib:/lib
```

3. Add the following line to the TUXEDO file udataobj/RM:

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: -lmqmxa -lmqm
```

4. Run the commands:

```
$ mkfldhdr    MQ_INSTALLATION_PATH/samp/amqstxvx.flds
$ viewc       MQ_INSTALLATION_PATH/samp/amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f MQ_INSTALLATION_PATH/samp/amqstxsx.c \
       -f MQ_INSTALLATION_PATH/lib/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT1:MPUT \
       -s MGET1:MGET \
       -v -bshm
$ buildserver -o MQSERV2 -f MQ_INSTALLATION_PATH/samp/amqstxsx.c \
       -f MQ_INSTALLATION_PATH/lib/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT2:MPUT
       -s MGET2:MGET \
       -v -bshm
$ buildclient -o doputs -f MQ_INSTALLATION_PATH/samp/amqstxpx.c \
       -f MQ_INSTALLATION_PATH/lib/libmqm.a
$ buildclient -o dogets -f MQ_INSTALLATION_PATH/samp/amqstxgx.c \
       -f MQ_INSTALLATION_PATH/lib/libmqm.a
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf  -y MQ_INSTALLATION_PATH/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt is displayed. At this prompt, enter:

```
> crdl -z /APPDIR/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

What to do next

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

AIX *Building the server environment for AIX (64-bit)*

How to build the server environment for IBM MQ for AIX (64-bit).

Procedure

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 represents the root directory for TUXEDO, and *MQ_INSTALLATION_PATH* represents the high-level directory in which IBM MQ is installed.:

```
$ export CFLAGS="-I MQ_INSTALLATION_PATH/inc -I /APPDIR -L MQ_INSTALLATION_PATH/lib64"
$ export LDOPTS="-lmqm"
$ export FIELDTBLS= MQ_INSTALLATION_PATH/samp/amqstxvx.flds
$ export VIEWFILES=/APPDIR/amqstxvx.V
$ export LIBPATH=$TUXDIR/lib64: MQ_INSTALLATION_PATH/lib64:/lib64
```

3. Add the following line to the TUXEDO file udataobj/RM:

```
MQSeries_XA_RMI:MQRMIXASwitchDynamic: -lmqmxa64 -lmqm
```

4. Run the commands:

```
$ mkfldhdr    MQ_INSTALLATION_PATH/samp/amqstxvx.flds
$ viewc       MQ_INSTALLATION_PATH/samp/amqstxvx.v
$ buildtms    -o MQXA -r MQSeries_XA_RMI
$ buildserver -o MQSERV1 -f MQ_INSTALLATION_PATH/samp/amqstxsx.c \
       -f MQ_INSTALLATION_PATH/lib64/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT1:MPUT \
       -s MGET1:MGET \
       -v -bshm
$ buildserver -o MQSERV2 -f MQ_INSTALLATION_PATH/samp/amqstxsx.c \
       -f MQ_INSTALLATION_PATH/lib64/libmqm.a \
       -r MQSeries_XA_RMI -s MPUT2:MPUT
       -s MGET2:MGET \
       -v -bshm
$ buildclient -o doputs -f MQ_INSTALLATION_PATH/samp/amqstxpx.c \
       -f MQ_INSTALLATION_PATH/lib64/libmqm.a
$ buildclient -o dogets -f MQ_INSTALLATION_PATH/samp/amqstxgx.c \
       -f MQ_INSTALLATION_PATH/lib64/libmqm.a
```

5. Edit ubbstxcx.cfg and add details of the machine name, working directories, and queue manager as necessary:

```
$ tmloadcf  -y MQ_INSTALLATION_PATH/samp/ubbstxcx.cfg
```

6. Create the TLOGDEVICE:

```
$tmadmin -c
```

   A prompt is displayed. At this prompt, enter:

```
> crdl -z /APPDIR/TLOG1
```

7. Start the queue manager:

```
$ strmqm
```

8. Start Tuxedo:

```
$ tmboot -y
```

What to do next

You can now use the doputs and dogets programs to put messages to a queue and retrieve them from a queue.

Windows *Building the server environment for Windows (32-bit)*
Building the server environment for IBM MQ for Windows (32-bit).

About this task

Note: Change the fields identified as *VARIABLES* in the following, to the directory paths:

Table 164. Fields to change to directory paths

Field	Directory path
MQMDIR	The directory path specified when IBM MQ was installed, for example `g:\Program Files\IBM\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`.

```
*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 134. Example of ubbstxcn.cfg file for IBM MQ for Windows

Note: Change the machine name *MachineName*, and the 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.

The sample `ubbconfig` file for IBM MQ for Windows is listed in Figure 134 on page 1086. It is supplied as `ubbstxcn.cfg` in the IBM MQ samples directory.

The sample makefile (see Figure 135 on page 1087) supplied for IBM MQ for Windows is called `ubbstxmn.mak`, and is held in the IBM 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 135. Sample TUXEDO makefile for IBM MQ for Windows

To build the server environment and samples, complete the following steps.

Procedure

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 IBM 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 134 on page 1086) 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
   ```

   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 following command:

   ```
   crdl -z APPDIR\TLOG
   ```

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 following command:

   ```
   msvc amqstxmn.mak
   ```

   Select **build** to build all the sample programs.

Windows *Building the server environment for Windows (64-bit)*

How to build the server environment for IBM MQ for Windows (64-bit).

About this task

Note: Change the fields identified as *VARIABLES* in the following, to the directory paths:

Table 165. Fields to change to directory paths

Field	Directory path
MQMDIR	The directory path specified when IBM MQ was installed, for example `g:\Program Files\IBM\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`.

```
*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:\Programï¿½Files\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 136. Example of ubbstxcn.cfg file for IBM MQ for Windows

Note: Change the machine name *MachineName*, and the 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.

The sample ubbconfig file for IBM MQ for Windows is listed in Figure 136 on page 1089. It is supplied as `ubbstxcn.cfg` in the IBM MQ samples directory.

The sample makefile (see Figure 137 on page 1090) supplied for IBM MQ for Windows is called `ubbstxmn.mak`, and is held in the IBM 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 137. Sample TUXEDO makefile for IBM MQ for Windows

To build the server environment and samples, complete the following steps.

Procedure

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 IBM 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 136 on page 1089) 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
   ```

   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
   ```

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 following command:

```
msvc amqstxmn.mak
```

Select **build** to build all the sample programs.

ALW *Sample server program for TUXEDO*

The sample server program (amqstxsx) is designed to run with the Put (amqstxpx.c) and the Get (amqstxgx.c) sample programs. The sample server 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.

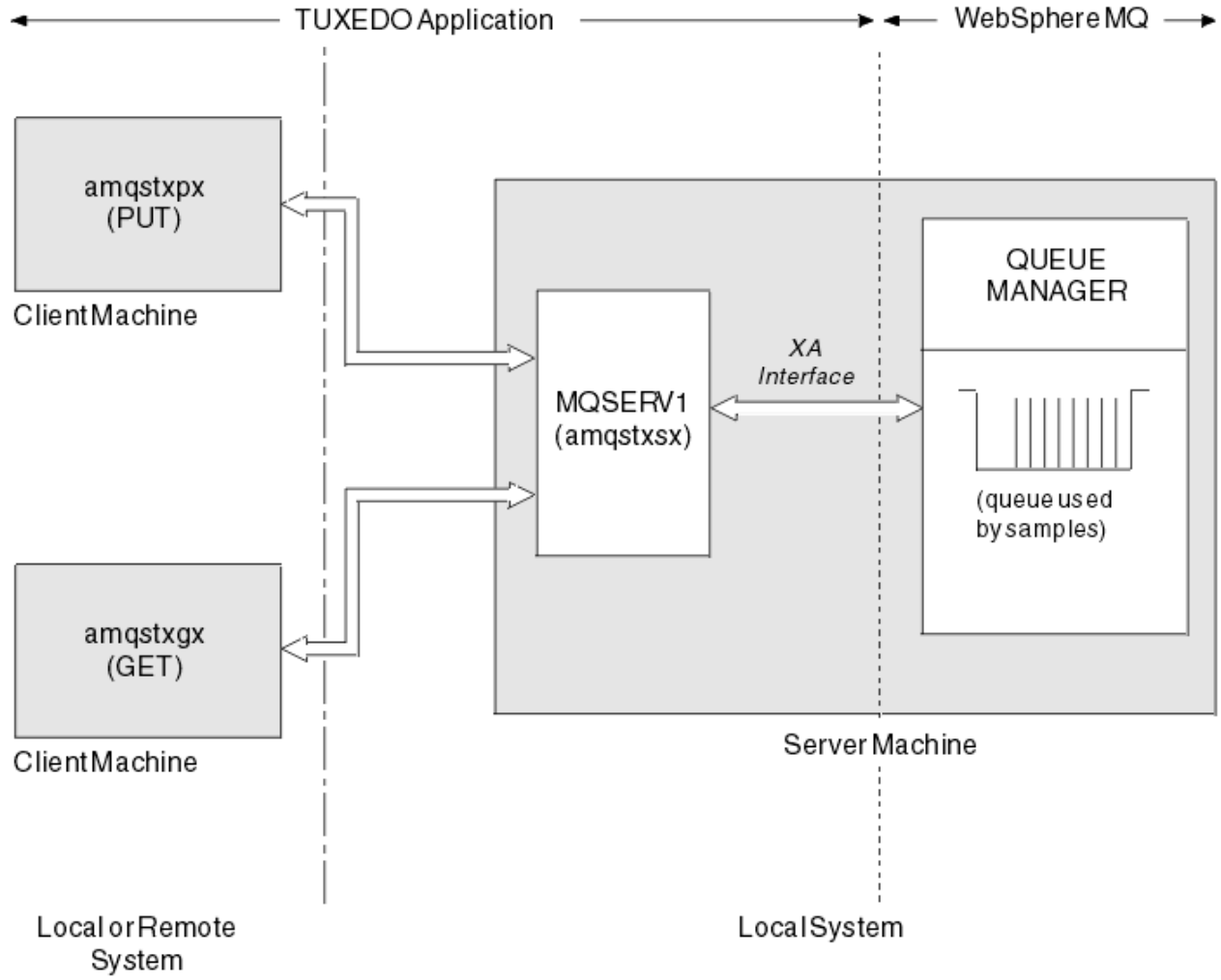


Figure 138. How TUXEDO samples work together

ALW *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.

ALW *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 Get sample to succeed; the sample server program connects to the queue manager and uses the XA interface. To run the sample, enter the following command:

- `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.

Windows *Using the SSPI security exit on Windows*

This topic 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 IBM MQ for Windows in the *MQ_INSTALLATION_PATH*`/exits/`*INSTALLATION_NAME* folder. For example, `C:\Program Files\IBM\MQ\exits\installation2`. It 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 SVRCONN 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\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 /Ipath_to_Microsoft_platform_SDK\include
/Ipath_to_IBM_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.

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 Defining the channels.

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.

The Cluster Queue Monitoring sample program (AMQSCLM)

This sample uses the built-in IBM MQ cluster workload balancing features to direct messages to instances of queues that have consuming applications attached. This automatic direction prevents the build-up of messages on an instance of a cluster queue to which no consuming application is attached.

Overview

You can set up a cluster that has more than one definition for the same queue on different queue managers. This configuration provides the benefit of increased availability and workload balancing. However, there is no capability built into IBM MQ to dynamically modify the distribution of messages across a cluster based on the state of attached applications. For this reason, a consuming application must always be attached to every instance of a queue to ensure that messages are processed.

The cluster queue monitoring sample program monitors the state of attached applications. The program dynamically adjusts the built-in workload balancing configuration to direct messages to instances of a clustered queue with consuming applications attached. In certain situations this program can be used to relax the need for a consuming application to always be connected to every instance of a queue. It also resends messages that become queued on an instance of a queue with no consuming applications attached. Resending messages enables messages to be routed around a consuming application that is temporarily shut down.

The program is designed to be used where the consuming applications are long running applications, rather than frequently attaching and detaching applications.

The cluster queue monitoring sample program is the compiled executable program of the C sample file `amqsclma.c`.

Further information about clusters and workload can be found in Using clusters for workload management

AMQSCLM: Design and Planning for using the sample
Information about how the cluster queue monitoring sample program works, points to consider when setting up a system for the sample program to run on, and modifications that can be made to the sample source code.

Design

The cluster queue monitoring sample program monitors local clustered queues that have consuming applications attached. The program monitors queues specified by the user. The name of the queue might be specific, for example `APP.TEST01`, or generic. Generic names must be in a format that conforms to PCF (Programmable Command Format). Examples of generic names are `APP.TEST*`, or `APP*`.

Each queue manager in a cluster that owns an instance of a local queue to be monitored, requires an instance of the cluster queue monitoring sample program to be connected to it.

Dynamic message routing

The cluster queue monitoring sample program uses the **IPPROCS** (open for input process count) value of a queue to determine whether that queue has any consumers. A value greater than 0 indicates that the queue has at least one consuming application attached. Such queues are active. A value of 0 indicates that the queue has no attached consuming programs. Such queues are inactive.

For a clustered queue with multiple instances in a cluster, IBM MQ uses the cluster workload priority property **CLWLPRTY** of each queue instance to determine which instances to send messages to. IBM MQ sends messages to the available instances of a queue with the highest **CLWLPRTY** value.

The cluster queue monitoring sample program activates a cluster queue by setting the local **CLWLPRTY** value to 1. The program deactivates a cluster queue by setting its **CLWLPRTY** value to 0.

IBM MQ clustering technology propagates the updated **CLWLPRTY** property of a clustered queue to all relevant queue managers in the cluster. For example,

- A queue manager with a connected application that puts messages to the queue.
- A queue manager that owns a local queue of the same name in the same cluster.

The propagation is done using the full repository queue managers of the cluster. New messages for the cluster queue are directed to the instances with the highest **CLWLPRTY** value within the cluster.

Queued message transfer

The dynamic modification of the value of **CLWLPRTY** influences the routing of new messages. This dynamic modification does not affect messages already queued on a queue instance with no attached consumers, or messages that have been through the workload balancing mechanism before a modified **CLWLPRTY** value was propagated across the cluster. As a result, messages remain on any inactive queue and not be processed by a consuming application. To solve this, the cluster queue monitoring sample program is able to get messages from a local queue with no consumers, and send these messages to remote instances of the same queue where consumers are attached.

The cluster queue monitoring sample program transfers messages from an inactive local queue to one or more active remote queues by getting messages (using **MQGET**) and putting messages (using **MQPUT**) to the same clustered queue. This transfer causes the IBM MQ cluster workload management to select a different target instance, based on a higher CLWLPRTY value than that of the local queue instance. Message persistence and context are preserved during the message transfer. Message order, and any binding options are not preserved.

Planning

The cluster queue monitoring sample program modifies the cluster configuration when there is a change in the connectivity of consuming applications. Modifications are transmitted from the queue managers where the cluster queue monitoring sample program is monitoring queues, to the full repository queue managers in the cluster. The full repository queue managers process the configuration updates and resend them to all relevant queue managers in the cluster. Relevant queue managers include those queue managers that own clustered queues of the same name (where an instance of the cluster queue monitoring sample program is running), and any queue manager where an application opened the cluster queue to put messages to it in the last 30 days.

Changes are asynchronously processed across the cluster. Therefore, after each change, different queue managers in the cluster might have different views of the configuration for a period of time.

The cluster queue monitoring sample program is only suitable for systems where consuming applications infrequently attach or detach; for example, long running consuming applications. When used to monitor systems where consuming applications are only attached for short periods, the latency incurred when distributing the configuration updates might result in queue managers in the cluster having an incorrect view of the queues where consumers are attached. This latency might result in incorrectly routed messages.

When monitoring many queues, a relatively low rate of change in attached consumers across all queues might increase cluster configuration traffic across the cluster. Increased cluster configuration traffic can result in excessive load on one or more of the following queue managers.

- The queue managers where the cluster queue monitoring sample program is running
- The full repository queue managers
- A queue manager with a connected application that puts messages to the queue
- A queue manager that owns a local queue of the same name in the same cluster

Processor usage on the full repository queue managers must be assessed. Additional processor usage is visible as message traffic on the full repository queue SYSTEM.CLUSTER.COMMAND.QUEUE. If messages build up on that queue, it indicates that the full repository queue managers are unable to keep up with the rate of cluster configuration change in the system.

When many queues are being monitored by the cluster queue monitoring sample program, there is an amount of work performed by the sample program and the queue manager. This work is performed, even when there are no changes to the attached consumers. The **-i** argument can be modified to reduce processor usage of the sample program on the local system, by decreasing the frequency of the monitoring cycle.

To help detect excessive activity, the cluster queue monitoring sample program reports average processing time per polling interval, elapsed processing time, and number of configuration changes. The reports are delivered in an information message, **CLM0045I**, every 30 minutes, or every 600 poll intervals, whichever is sooner.

Cluster queue monitoring usage requirements

The cluster queue monitoring sample program has requirements and restrictions. You can modify the sample source code provided to change some of these restrictions in how it can be used. Examples listed in this section detail modifications that can be made.

- The cluster queue monitoring sample program is designed to be used to monitor queues where consuming applications are either attached, or not attached. If the system has consuming applications that are frequently attaching and detaching, the sample program might generate excessive cluster configuration activity across the entire cluster. This might have an impact on the performance of the queue managers in the cluster.
- The cluster queue monitoring sample program depends upon the underlying IBM MQ system and cluster technology. The number of queues being monitored, the frequency of monitoring and the frequency of the change of the state of each queue affects the load on the overall system. These factors must be considered when selecting the queues to be monitored and the poll interval of the monitoring.

- An instance of the cluster queue monitoring sample program must be connected to every queue manager in the cluster that owns an instance of a queue to be monitored. It is not necessary to connect the sample program to queue managers in the cluster that do not own the queues.
- The cluster queue monitoring sample program must be run with suitable authorization to access all of the IBM MQ resources required. For example,
  - The queue manager to be connected to
  - The SYSTEM.ADMIN.COMMAND.QUEUE
  - All the queues to be monitored when message transfer is performed
- The command server must be running for each queue manager with the cluster queue monitoring sample program connected.
- Each instance of the cluster queue monitoring sample program requires exclusive use of a local (non-clustered) queue on the queue manager that it is connected to. This local queue is used to control the sample program, and receive reply messages from inquires made to the command server of the queue manager.
- All queues to be monitored by a single instance of the cluster queue monitoring sample program must be in the same cluster. If a queue manager has queues in multiple clusters that require monitoring, multiple instances of the sample program are required. Each instance needs a local queue for control and reply messages.
- All queues to be monitored must be in a single cluster. Queues configured to use a cluster namelist are not monitored.
- Enabling the transfer of messages from inactive queues is optional. It applies to all queues being monitored by the instance of the cluster queue monitoring sample program. If only a subset of the queues being monitored require message transfer enabled, two instances of the cluster queue monitoring sample program are needed. One sample program has message transfer enabled, and the other has message transfer disabled. Each instance of the sample program needs a local queue for control and reply messages.
- IBM MQ cluster workload balancing will, by default, send messages to instances of clustered queues that reside on the same queue manager that a putting application is connected to. This must be disabled while the local queue is inactive in the following circumstances:
  - Putting applications connect to queue managers that own instances of an inactive queue that are being monitored
  - Queued messages are being transferred from inactive queues to active queues.

  The local workload balancing preference on the queue can be disabled statically, through setting the `CLWLUSEQ` value to `ANY`. In this configuration messages put on local queues are distributed to local and remote queue instances to balance workload, even when there are local consuming applications. Alternatively, the cluster queue monitoring sample program can be configured to temporarily set the **CLWLUSEQ** value to `ANY` while the queue has no attached consumers which results in only local messages going to local instances of a queue while that queue is active.
- The IBM MQ system and applications must not use **CLWLPRTY** for the queues to be monitored, or channels being used. Otherwise, the actions of the cluster queue monitoring sample program on **CLWLPRTY** queue attributes might have undesired effects.
- The cluster queue monitoring sample program logs runtime information to a set of report files. A directory to store these reports is required, and the cluster queue monitoring sample program must have authorization to write to it.

AMQSCLM: Preparing and running the sample

The cluster queue monitoring sample can either be run locally connected to a queue manager, or as a client connected over a channel. The sample should be running whenever the queue manager is running,

when running locally it can be configured as a queue manager service to automatically start and stop the sample with the queue manager.

Before you begin

The following steps must be completed before running the cluster queue monitoring sample.

1. Create a working queue on each queue manager for the internal use of the sample.

   Each instance of the sample needs a local non-cluster queue for exclusive internal use. You can choose the name of the queue. The example uses the name `AMQSCLM.CONTROL.QUEUE`. For example, on Windows, you can create this queue by using the following **MQSC** command:

   ```
   DEFINE QLOCAL(AMQSCLM.CONTROL.QUEUE)
   ```

   You can leave the values of **MAXDEPTH** and **MAXMSGL** as default.

2. Create a directory for error and information message logs.

   The sample writes diagnostic messages to report files. You must choose a directory in which to store the files. For example, on Windows, you can create a directory using the following command:

   ```
   mkdir C:\AMQSCLM\rpts
   ```

   The report files created by the sample have the following naming convention:

   ```
   QmgrName.ClusterName.RPT0n.LOG
   ```

3. (Optional) Define the cluster queue monitoring sample as an IBM MQ service.

   To monitor queues, the sample must always be running. To ensure that the cluster queue monitoring sample is always running, you can define the sample as a queue manager service. Defining the sample as a service means that AMQSCLM is started when the queue manager starts. You can use the following example to define the cluster queue monitoring sample as an IBM MQ service.

   ```
   define service(AMQSCLM) +
         descr('Active Cluster Queue Message Distribution Monitor - AMQSCLM') +
         control(qmgr) +
         servtype(server) +
         startcmd('MQ_INSTALLATION_PATH\tools\c\samples\Bin\AMQSCLM.exe') +
         startarg('-m +QMNAME+ -c CLUSTER1 -q ABC* -r AMQSCLM.CONTROL.QUEUE -l
   c:\AMQSCLM\rpts') +
         stdout('C:\AMQSCLM\rpts\+QMNAME+.TSTCLUS.stdout.log') +
         stderr('C:\AMQSCLM\rpts\+QMNAME+.TSTCLUS.stderr.log')
   ```

Definition	Description
service	Specifies the service name. You can choose the service name.
descr	Specifies a textual description of the service.
control	Indicates that the service starts and stops at the same time as the queue manager.
servtype	Indicates a server service object, meaning only one instance, can be executed at a time for this queue manager.
startcmd	Specifies the location and name of the program.
startarg	Specifies the arguments of the sample. Note the use of *+QMNAME+*. The name of the queue manager is automatically substituted.
stdout	The fully qualified file name to which standard output is redirected. The sample writes to this file only messages confirming that the sample has terminated. The sample does

Definition	Description
	this because the standard error file has already closed in an earlier stage of the sample termination process.
stderr	The fully qualified file name to which standard error output is redirected. The sample writes to the standard error file any error messages prior to termination of the sample.

About this task

This task enables you to start and stop the cluster queue monitoring sample in different ways. It also enables you to run the sample in a mode that generates report files containing statistical information about the queues being monitored.

The sample program can be run by using the following command.

```
AMQSCLM -m QMgrName -c ClusterName (-q QNameMask| -f QListFile) -r MonitorQName
[-l ReportDir] [-t] [-u ActiveVal] [-i Interval] [-d] [-s] [-v]
```

The table lists the arguments that can be used with the cluster queue monitoring sample, along with additional information about each.

Argument	Variable	Further Information
-m	**QMgrName**	The queue manager to monitor.
-c	**ClusterName**	The cluster containing the queues to monitor.
-q	**QNameMask**	The queue, or queues, to monitor. A trailing * monitors all queues with names that match zero or more trailing characters.
-f	**QListFile**	The full path and file name of a file containing a list of queue names or queue name masks to monitor. The file must contain one queue name/mask per line. You can specify **-q** or **-f**, but not both.
-r	**MonitorQName**	The local queue being used exclusively by the sample.
-l	**ReportDir**	The directory path in which to store logged information messages in a set of wrapping[9] report files.
-t		(Optional) Enables the transfer of queued messages from inactive local queues to active queues. If not enabled, only new messages entering the cluster are dynamically routed to active instances of a queue.
-u	**ActiveVal**	(Optional) Automatically switches the **CLWLUSEQ** property of a monitored queue instance to ANY when it is inactive, and to the value of **ActiveVal** when active. **ActiveVal** can be LOCAL or QMGR. If this argument is not set in a system where putting applications connect to the same queue manager, or where message transfer is enabled, then monitored queues must have a **CLWLUSEQ** value of ANY, or QMGR with the queue manager having a value of ANY.
-i	**Interval**	(Optional) The time interval in seconds, at which the monitor checks the queues. Default is 300 seconds (5 minutes).
-d		(Optional) Enables additional diagnostic output. Debug output might be useful when initially configuring the system, or when working with the sample code.
-s		(Optional) Enables minimal statistical output per interval.
-v		(Optional) Log report information to `standard out`, in addition to the report files.

Argument list examples:

```
-m QMGR1 -c CLUS1   -f c:\QList.txt -r CLMQ -l c:\amqsclm\rpts -s
-m QMGR2 -c CLUS1   -q ABC* -r CLMQ -l c:\amqsclm\rpts -i 600
-m QMGR1 -c CLUSDEV -q QUEUE.* -r CLMQ -l c:\amqsclm\rpts -t -u QMGR -d
```

Example queue list file:

```
Q1
QUEUE.*
ABC
ABD
```

[9] For each queue manager and queue combination a fixed-size log file is generated that, when full, is overwritten. The logger always writes into the same file, and also keeps the two previous versions of the file.

Procedure

1. Start the cluster queue monitoring sample. You can start the sample in one of the following ways:
   - Use a command prompt with the appropriate user authorizations.
   - Use the MQSC **START SERVICE** command, if the sample is configured as an IBM MQ service.

   The argument list is the same in both cases.

   The sample does not start monitoring the queues for 10 seconds after the program is initialized. This delay allows consuming applications to connect to the monitored queues first, preventing unnecessary changes to the active state of the queue.
2. Stop the cluster queue monitoring sample. The sample automatically stops when the queue manager is stopped, stopping, quiescing, or if the connection to the queue manager is broken. There are ways to stop the sample without ending the queue manager:
   - Configure the local queue used exclusively by the sample to disable the Get function.
   - Send a message with a **CorrelId** of `"STOP CLUSTER MONITOR\0\0\0\0"`, to the local queue used exclusively by the sample.
   - Terminate the sample process. This might result in the loss of non-persistent messages being transferred to active queues. It might also result in the local queue used by the sample being held open for a number of seconds after the termination. This situation prevents a new instance of the cluster queue monitoring sample from starting immediately.

   If the sample has been started as an IBM MQ service, **STOP SERVICE** has no effect. It is possible to use one of the termination methods described as a configured **STOP SERVICE** mechanism in the queue manager.

What to do next

Check the status of the sample.

If reporting is enabled, you can review the report files for status. Use the following command to review the most current report file:

```
QMgrName.ClusterName.RPT01.LOG
```

To review older report files, use the following commands:

```
QMgrName.ClusterName.RPT02.LOG
QMgrName.ClusterName.RPT03.LOG
```

Report files grow to a maximum size of approximately 1 MB. When the RPT01 file fills up, a new RPT01 file is created. The old RPT01 file is renamed to RPT02. RPT02 is renamed to RPT03. The old RPT03 is discarded.

The sample creates information messages in the following situations:

- at startup
- at termination
- when it marks a queue **ACTIVE** or **INACTIVE**
- when it requeues messages from an inactive queue to an active instance or instances

The sample creates an error message *CLMnnnnE* to report a problem that requires attention.

Every 30 minutes, the sample reports average processing time per polling interval, and elapsed processing time. This information is held in message CLM0045I.

When statistical messages are enabled **-s**, the sample reports the following statistical information about each queue check:

- Time taken to process the queues (in milliseconds)

- Number of queues checked
- Number of active/inactive changes made
- Number of messages transferred

This information is reported in message CLM0048I.

Report files might grow rapidly in debug mode, and quickly wrap. In this situation, the 1 MB size limit for individual files might be exceeded.

AMQSCLM: Troubleshooting
The following sections contain information about scenarios that might be encountered while using the sample. Information about potential explanations for a scenario, and options on how to resolve it, are provided.

Scenario: AMQSCLM is not starting

Potential explanation: Incorrect syntax.

Action: Check standard error output for correct syntax

Potential explanation: Queue manager is not available.

Action: Check the report file for message ID CLM0010E.

Potential explanation: Cannot open or create report file or files.

Action: Check standard error output for error messages during initialization.

Scenario: AMQSCLM is not changing a queue to ACTIVE or INACTIVE

Potential explanation: The queue is not in the list of queues to be monitored

Action: Check the **-q** and **-f** parameter values.

Potential explanation: The queue is not a local queue in the correct cluster.

Action: Check that the queue is local and in the correct cluster.

Potential explanation: AMQSCLM is not running for this queue manager and cluster.

Action: Start AMQSCLM for the relevant queue manager and cluster.

Potential explanation: The queue is left INACTIVE, **CLWLPRTY** =0, because it has no consumers. Alternatively, it is left ACTIVE **CLWLPRTY** >=1, because it has at least 1 consumer.

Action: Check whether consuming applications are attached to the queue.

Potential explanation: The command server of the queue manager is not running.

Action: Check the report files for errors.

Scenario: Messages are not being routed around INACTIVE queues

Potential explanation: Messages are put directly to the queue manager that owns the inactive queue, and the **CLWLUSEQ** value of the queue is not ANY, and the **-u** argument is not being used for AMQSCLM.

Action: Check the **CLWLUSEQ** value of the relevant queue manager, or ensure that the **-u** argument is used for AMQSCLM.

Potential explanation: There are no active queues on any queue managers. Messages are evenly workload balanced across all inactive queues until a queue becomes active.

Action: Check the status of the queues on all queue managers.

Potential explanation: Messages are put to a different queue manager in the cluster to the one that owns the inactive queue, and the updated **CLWLPRTY** value of 0 is not propagated to the queue manager of the putting application.

Action: Check that the cluster channels between the monitored queue manager, and the full repository queue manager, are running. Check that the channels between the putting queue manager, and the full repository queue manager, are running. Check the error logs of the monitored, putting, and full repository queue managers.

Potential explanation: The remote queue instances are active (CLWLPRTY=1), but messages cannot be routed to those queue instances because the cluster sender channel from the local queue manager is not running.

Action: Check the status of the cluster sender channels from the local queue manager to the remote queue manager, or managers, with an active instance of the queue.

Scenario: AMQSCLM is not transferring messages from an inactive queue

Potential explanation: Message transfer is not enabled (**-t**).

Action: Ensure that message transfer is enabled (**-t**).

Potential explanation: The queue is not in the list of queues to be monitored.

Action: Check the **-q** and **-f** parameter values.

Potential explanation: AMQSCLM is not running for this, or other queue managers in the cluster, that own instances of the same queue.

Action: Start AMQSCLM.

Potential explanation: The queue has **CLWLUSEQ** = LOCAL or **CLWLUSEQ** = QMGR, and the **-u** argument is not set.

Action: Set the **-u** parameter, or change the queue, or queue manager configuration, to ANY.

Potential explanation: There are no active instances of the queue in the cluster.

Action: Check for instances of the queue with a **CLWLPRTY** value of 1, or greater.

Potential explanation: Remote queue instances have consumers (**IPPROCS** >=1) but are inactive on those queue managers (**CLWLPRTY** =0) because AMQSCLM is not monitoring those remote instances.

Action: Ensure that AMQSCLM is running on those queue managers, and / or the queue is in the list of queues to be monitored by checking the **-q** and **-f** parameter values.

Potential explanation: The remote queue instances are active (**CLWLPRTY** =1), but are seen as inactive on the local queue manager (**CLWLPRTY** =0). This situation is due to the updated **CLWLPRTY** value not being propagated to this queue manager.

Action: Ensure that the remote queue managers are connected to at least one of the full repository queue managers in the cluster. Ensure that the full repository queue managers are functioning correctly. Check that the channels between the full repository queue managers, and the monitored queue managers, are running.

Potential explanation: The messages are not committed, therefore they are not retrievable.

Action: Check that the sending application is functioning correctly.

Potential explanation: AMQSCLM does not have access to the local queue where messages are queued.

Action: Check whether AMQSCLM is running as a user with sufficient authorization to access the queue.

Potential explanation: The command server of the queue manager is not running.

Action: Start the command server of the queue manager.

Potential explanation: AMQSCLM encountered an error.

Action: Check the report files for errors.

Potential explanation: The remote queue instances are active (CLWLPRTY=1), but messages cannot be transferred to those queue instances because the cluster sender channel from the local queue manager is not running. This is often accompanied by a CLM0030W warning in the amqsclm report log.

Action: Check the status of the cluster sender channels from the local queue manager to the remote queue manager, or managers, with an active instance of the queue.

ALW *Sample program for Connection Endpoint Lookup (CEPL)*

IBM MQ Connection Endpoint Lookup sample provides a simple yet powerful exit module that offers IBM MQ users a way to retrieve connection definitions from an LDAP repository such as Tivoli Directory Server.

Tivoli Directory Server v6.3 Client must be installed in order to use CEPL.

A working knowledge of IBM MQ administration on the supported platforms is required to use this sample.

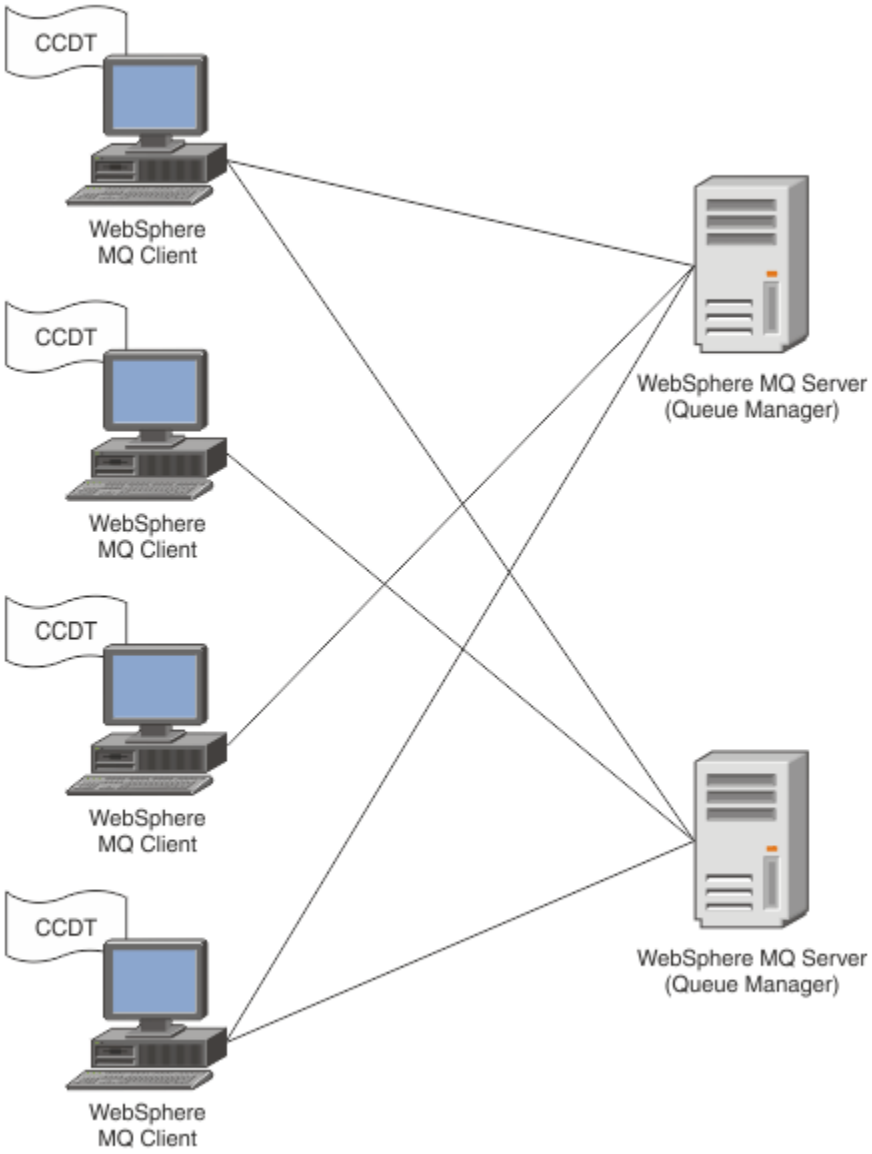
Windows Linux AIX *Introduction*

Configure a global repository, for example, an LDAP (Lightweight Directory Access Protocol) directory, to store client connection definitions to aid maintenance and administration.

Using an IBM MQ Client application to establish a connection to a Queue Manager via a Client Connection Definition Table (CCDT).

The CCDT is created through the standard IBM MQ MQSC Administration interface. The user must be connected to a Queue Manager in order to create client connection definitions, even though the data contained within the definition is not restricted to the Queue Manager. The

CCDT file generated must be manually distributed among client machines and applications.

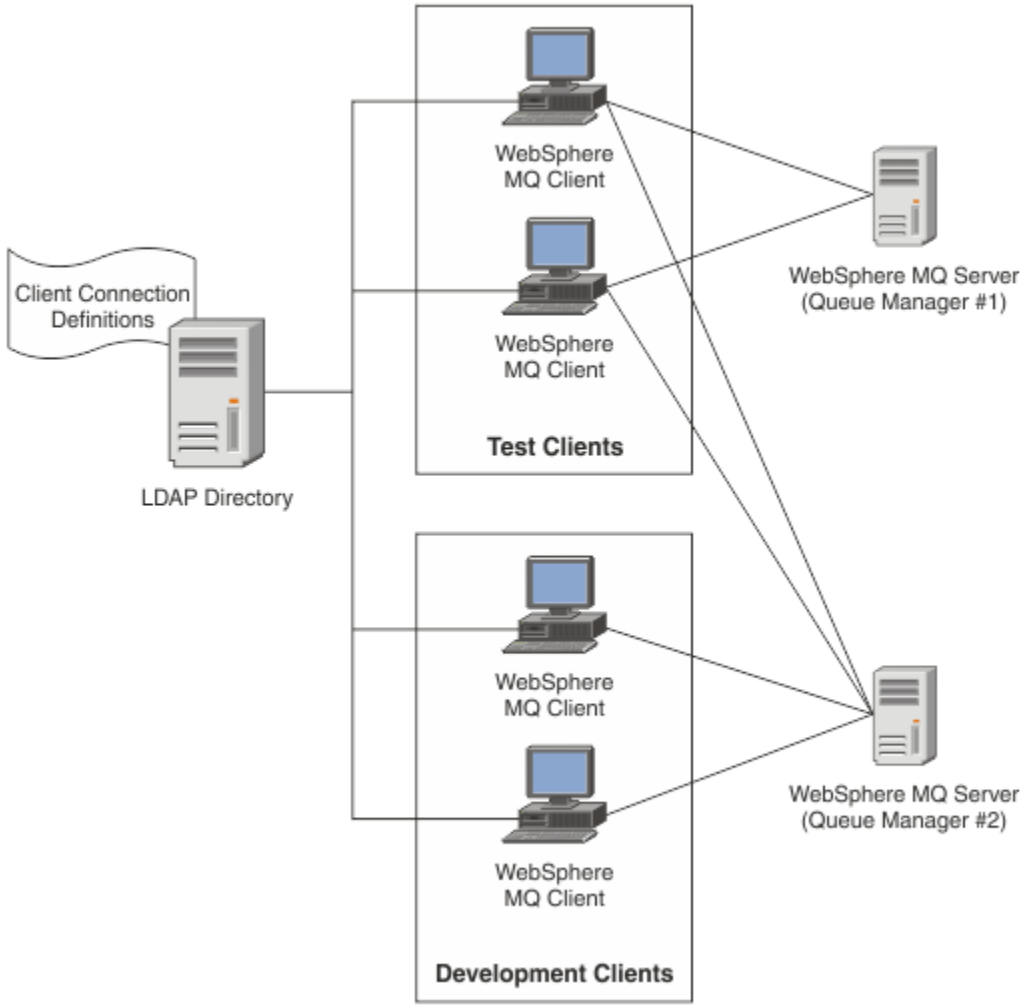


The CCDT file must be distributed to each IBM MQ client. Where thousands of clients can exist either locally or globally, it would soon become difficult to maintain and administer. A more flexible approach is needed to help ensure that each client has the correct client definitions available to them.

One such approach is to store the client connection definitions in a global repository such as an LDAP (Lightweight Directory Access Protocol) directory. An LDAP directory can also provide additional security, indexing, and search facilities, thereby allowing each client access to only those connection definitions pertaining to them.

The LDAP directory can be configured so that only specific definitions are available to certain user groups. For example, the Test Clients can access both Queue Manager

#1 and #2, whereas the Development Clients can access Queue Manager #2 only.



The exit module can look up an LDAP repository, for example, IBM Tivoli Directory Server, to retrieve channel definitions. Using those connection definitions, an IBM MQ client application can establish connection to a queue manager.

The exit module is a pre-connect exit module which enables channel definition to be obtained during the MQCONN/MQCONNX call from an LDAP repository.

The exit module and schema might be implemented by:

- Customers who have already built a skill base using the existing CCDT file based technology and want to ease the administration and distribution costs.
- Existing customers who already employ their own propriety technology for distributing client connection definitions.
- New or existing customers who currently do not employ any type of client connection solution and want to use the features offered by IBM MQ.
- New or existing customers who want to directly use or tune their messaging model inline with any current LDAP business architecture.

ALW *Supported environments*

Verify that you have a supported operating system and the relevant software before running the Connection Endpoint Lookup sample.

The sample program for IBM MQ Connection Endpoint Lookup requires the following software:

- IBM WebSphere MQ 7.0, or later

- Tivoli Directory Server V6.3 Client, or later

Operating Systems supported:

1. Windows (7/8/2008/2012)
2. AIX
3. Linux
   - RHEL v4 and v5 on System p
   - SUSE v9 and v10 on System p
   - RHEL v4 and v5 x86-64 32 bit and 64 bit
   - SUSE v9 and v10 x86-64 32 bit and 64 bit

Note: The sample is not available for the following platforms:

- z/OS
- IBM i

ALW *Installation and configuration*
Installing and configuring the exit module and connection endpoint schema.

Installing exit module

During installation of IBM MQ, the exit module is installed under `tools/samples/c/preconnexit/bin`. For 32 bit platforms, the exit module must be copied to `exit/`*`installation_name`*`/` before it can be used. For 64 bit platforms, the exit module must be copied to exit64/*installation_name*/ before it can be used.

Installing Connection Endpoint schema

The exit uses the Connection Endpoint schema, `ibm-amq.schema`. The schema file must be imported into any LDAP server before the exit can be used. After importing the schema, values for the attributes must be added.

Here is an example for importing Connection Endpoint schema. The example assumes IBM Tivoli Directory Server (ITDS) is being used.

- Ensure that IBM Tivoli Directory Server is running, then copy or FTP the `ibm-amq.schema` file to the ITDS server.
- On the ITDS server, enter the following command to install the schema into ITDS store, where *LDAP ID* and *LDAP password* are the root DN and password for the LDAP server:

  `ldapadd -D "`*`LDAP ID`*`" -w "`*`LDAP password`*`" -f ibm-amq.schema`
- In a command window, enter the following command or use a third party tool to browse the schema for verification:

```
ldapsearch objectclass=ibm-amqClientConnection
```

Refer to your LDAP Server documentation for further details on importing the schema file.

Configuration

A new section called PreConnect must be added to the client configuration file, for example `mqclient.ini`. The PreConnect section contains the following keywords:

Module
The name of the module containing the API exit code. If this field contains the full path of the module, it is used as is. Otherwise the `exit` or `exit64` folder in the IBM MQ installation is searched.

Function
The name of the functional entry point into the library that contains the `LdapPreConnect` exit code. The function definition adheres to the function prototype of your enterprise.

Attention: You should remove the quotation marks in the function statement when you specify your actual exit entry point.

Data
URI of the LDAP repository containing channel definitions.

The following snippet is an example of the changes required in the `mqclient.ini` file.

```
PreConnect:
Module=amqlcelp
Function="LdapPreconnectExit"
Data=ldap:dap://myLDAPServer.com:389/cn=wmq,ou=ibm,ou=com
Sequence=1
```

ALW *Overview of the exit and schema*
Syntax, and parameters used to establish a connection to a queue manager.

IBM MQ 9.3 defines the following syntax for an entry point in an exit module.

```
void MQENTRY MQ_PRECONNECT_EXIT ( PMQNXP pExitParms
                                , PMQCHAR pQMgrName
                                , PPMQCNO ppConnectOpts
                                , PMQLONG pCompCode
                                , PMQLONG pReason)
```

During the MQCONN/X call execution, IBM MQ C Client loads the exit module containing an implementation of the function syntax. It then invokes an exit function to retrieve channel definitions. The retrieved channel definitions are then used to establish connection to a queue manager.

Parameters

pExitParms
Type: PMQNXP input/output

The PreConnection exit parameter structure. The structure is allocated and maintained by the caller of the exit.

```
struct tagMQNXP
{
  MQCHAR4    StrucId;             /* Structure identifier */
  MQLONG     Version;             /* Structure version number */
  MQLONG     ExitId;              /* Type of exit */
  MQLONG     ExitReason;          /* Reason for invoking exit */
  MQLONG     ExitResponse;        /* Response from exit */
  MQLONG     ExitResponse2;       /* Secondary response from exit */
  MQLONG     Feedback;            /* Feedback code (reserved) */
  MQLONG     ExitDataLength;      /* Exit data length */
  PMQCHAR    pExitDataPtr;        /* Exit data */
  MQPTR      pExitUserAreaPtr;    /* Exit user area */
  PMQCD *    ppMQCDArrayPtr;      /* Array of pointers to MQCDs */
  MQLONG     MQCDArrayCount;      /* Number of entries found */
  MQLONG     MaxMQCDVersion;      /* Maximum MQCD version */
};
```

pQMgrName
Type: PMQCHAR input/output

Name of the queue manager. On input, this parameter is the filter string supplied to the MQCONN API call through the **QMgrName** parameter. This field might be blank, explicit, or contain certain wildcard

characters. The field is changed by the exit. The parameter is NULL when the exit is called with MQXR_TERM.

ppConnectOpts

Type: ppConnectOpts input/output

Options that control the action of MQCONNX. This is a pointer to an MQCNO connection options structure that controls the action of the MQCONN API call. The parameter is NULL when the exit is called with MQXR_TERM. The MQI client always provides an MQCNO structure to the exit, even if it was not originally provided by the application. If an application provides an MQCNO structure, the client makes a duplicate to pass it to the exit where it is modified. The client retains the ownership of the MQCNO. An MQCD referenced through the MQCNO takes precedence over any connection definition provided through the array. The client uses the MQCNO structure to connect to the queue manager and the others are ignored.

pCompCode

Type: PMQLONG input/output

Completion code. Pointer to an MQLONG that receives the exits completion code. It must be one of the following values:

- MQCC_OK - Successful completion
- MQCC_WARNING - Warning (partial completion)
- MQCC_FAILED - Call failed

pReason

Type: PMQLONG input/output

Reason qualifying pCompCode. Pointer to an MQLONG that receives the exit reason 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.

ALW *MQ LDAP Context Information*

The exit uses the following data structure for context information.

MQNLDAPCTX

The MQNLDAPCTX structure has the following C prototype.

```
typedef struct tagMQNLDAPCTX MQNLDAPCTX;
typedef MQNLDAPCTX MQPOINTER PMQNLDAPCTX;

struct tagMQNLDAPCTX
{
  MQCHAR4      StrucId;          /* Structure identifier */
  MQLONG       Version;          /* Structure version number */
  LDAP *       objectDirectory   /* LDAP Instance */
  MQLONG       ldapVersion;      /* Which LDAP version to use? */
  MQLONG       port;             /* Port number for LDAP server*/
  MQLONG       sizeLimit;        /* Size limit */
  MQBOOL       ssl;              /* SSL enabled? */
  MQCHAR *     host;             /* Hostname of LDAP server */
  MQCHAR *     password;         /* Password of LDAP server */
  MQCHAR *     searchFilter;     /* LDAP search filter */
  MQCHAR *     baseDN;           /* Base Distinguished Name */
  MQCHAR *     charSet;          /* Character set */
};
```

Windows Linux AIX *Sample code for building the connection endpoint lookup exit*

You can use the sample code snippets for compiling the source on AIX, or Linux, or Windows.

Compiling source

You can compile the source with any LDAP client libraries, for example, IBM Tivoli Directory Server V6.3 Client libraries. This documentation assumes that you are using Tivoli Directory Server V6.3 client libraries.

Note: The pre-connect exit library is supported with the following LDAP servers:

- IBM Tivoli Directory Server V6.3
- Novell eDirectory V8.2

The following code snippets describe how to compile the exits:

Windows **Compiling the exit on the Windows platform**

You can use the following snippet for compiling the exit source:

```
CC=cl.exe
LL=link.exe
CCARGS=/c /I. /DWIN32 /W3 /DNDEBUG /EHsc /D_CRT_SECURE_NO_DEPRECATE /Zl

# The libraries to include
LDLIBS=Ws2_32.lib Advapi32.lib libibmldapstatic.lib libibmldapdbgstatic.lib \
kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib \
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib msvcrt.lib

OBJS=amqlcel0.obj

all: amqlcelp.dll

amqlcelp.dll: $(OBJS)
        $(LL) /OUT:amqlcelp.dll /INCREMENTAL /NOLOGO /DLL /SUBSYSTEM:WINDOWS /MACHINE: X86 \
              /DEF:amqlcelp.def $(OBJS) $(LDLIBS) /NODEFAULTLIB:msvcrt.lib

# The exit source
amqlcel0.obj: amqlcel0.c
$(CC) $(CCARGS) $*.c
```

Note: If you are using the IBM Tivoli Directory Server V6.3 Client libraries that are compiled with Microsoft Visual Studio 2003 compiler, you might get warnings while you are compiling the IBM Tivoli Directory Server V6.3 Client libraries with Microsoft Visual Studio 2012, or later, compiler.

Linux AIX **Compiling the exit on AIX, or Linux**

The following code snippet is for compiling the exit source on Linux. Some compiler options might differ on AIX.

```
#Make file to build exit
CC=gcc

MQML=/opt/mqm/lib
MQMI=/opt/mqm/inc
TDSI=/opt/ibm/ldap/V6.3/include
XFLAG=-m32

TDSL=/opt/ibm/ldap/V6.3/lib
```

IBM Tivoli Directory Server ships both static and dynamic link libraries, but you can use only one type of library. This script assumes that you are using the static libraries.

```
#Use static libraries.
LDLIBS=-L$(TDSL) -libibmldapstatic

CFLAGS=-I. -I$(MQMI) -I$(TDSI)

all:amqlcepl
```

```
amqlcepl: amqlcel0.c
$(CC) -o cepl amqlcel0.c -shared -fPIC $(XFLAG) $(CFLAGS) $(LDLIBS)
```

ALW *Invocation of the PreConnect exit module*

The PreConnect exit module can be invoked with three different reason codes: the MQXR_INIT reason code for initializing and establishing a connection to an LDAP server, the MQXR_PRECONNECT reason code for retrieving channel definitions from an LDAP server, or the MQXR_TERM reason code when the exit is to be cleaned.

MQXR_INIT

The exit is invoked with MQXR_INIT reason code for initializing and establishing a connection to an LDAP server.

Before the MQXR_INIT call, the pExitDataPtr field of the MQNXP structure is populated with the Data attribute from the PreConnect stanza within the `mqclient.ini` file (that is, the LDAP).

An LDAP URL consists of at least the protocol, host name, port number, and base DN for the search. The exit parses the LDAP URL contained within the pExitDataPtr field, allocates an MQNLDAPCTX LDAP Lookup Context structure and populates it accordingly. The address of this structure is stored in the pExitUserAreaPtr field. Failure to correctly parse the LDAP URL results in the error MQCC_FAILED.

At this point, the exit connects and binds to the LDAP server using the **MQNLDAPCTX** parameters. The resulting LDAP API handles are also stored within this structure.

MQXR_PRECONNECT

The exit module is invoked with the MQXR_PRECONNECT reason code for retrieving channel definitions from an LDAP server.

The exit searches the LDAP server for channel definitions matching the given filter. If the **QMgrNameparameter** contains a specific queue manager name, the search returns all channel definitions for which the **ibm-amqQueueManagerName** LDAP attribute value matches with the given queue manager name.

If the **QMgrName** parameter is '*' or ' '(blank), then the search returns all channel definitions for which the **ibm-amqIsClientDefault Connection** endpoint attribute is set to TRUE.

After a successful search, the exit prepares one or an array of MQCD definitions and returns back to the caller.

MQXR_TERM

The exit is invoked with this reason code when the exit is to be cleaned. During this cleaning, the exit disconnects from the LDAP server, and releases all the memory allocated and maintained by the exit, including the MQNLDAPCTX structure, the pointer array, and every MQCD it references. Any other fields are set to the default values. The **pQMgrName** and **ppConnectOpts** exit parameters are unused during an exit with the MQXR_TERM reason code and may be `NULL`.

Related reference

PreConnect stanza of the client configuration file

ALW *LDAP schemas*

Client connection data is stored in a global repository called the LDAP (Lightweight Directory Access Protocol) directory. An IBM MQ client uses an LDAP directory to obtain the connection definitions. The structure of the IBM MQ client connection definitions within the LDAP directory is known as the LDAP schema. An LDAP schema is the collection of attribute type definitions, object class definitions, and other information which a server uses to determine whether a filter or attribute value assertion matches against the attributes of an entry, and whether to permit, add, and modify operations.

Storing data in the LDAP directory

The client connection definitions are located under a specific branch within the directory tree known as the connection point. Like all other nodes within an LDAP directory, the connection point has a Distinguished Name (DN) associated with it. You can use this node as the starting point for any queries

you make on the directory. Use filtering when querying the LDAP directory to return a subset of client connection definitions. You can restrict access to sub-trees based on permissions granted in other parts of the directory tree - for example, to users, departments, or groups.

Defining your own attributes and classes

Store the client channel definition by modifying the LDAP schema. All LDAP data definitions require objects and attributes. The objects and attributes are identified by an object identifier (OID) number which uniquely identifies the object or attribute. All classes within an LDAP schema inherit either directly or indirectly from the top object. The client channel definition object contains the attributes of the top object. All LDAP data definitions require objects and attributes:

- Object definitions are collections of LDAP attributes.
- Attributes are LDAP data types.

The description of each attribute and how they map to the normal IBM MQ properties are described in LDAP attributes.

ALW *LDAP attributes*

LDAP attributes defined are specific to IBM MQ and maps directly to the client connection properties.

IBM MQ Client Channel Directory String Attributes

The character string attributes with their mapping to IBM MQ properties are listed in the following table. The attributes can hold values of directoryString (UTF-8 encoded Unicode, that is, a variable byte encoding system that includes IA5/ASCII as a subset) syntax. The syntax is specified by its object identification number (OID).

Table 166. IBM MQ client channel directory string attributes

LDAP Attribute	Description	IBM MQ Property
CN	The common name consisting of the channel name and the defining queue manager name.	
ibm-amqChannelName	The name of the channel definition.	CHANNEL
ibm-amqConnectionName	The communication connection identifier.	CONNAME
ibm-amqDescription	The channel description.	DESCR
ibm-amqLocalAddress	The local communication address of the channel.	LOCLADDR
ibm-amqModeName	The LU 6.2 mode name.	MODENAME
ibm-amqPassword	The password that can be used.	PASSWORD
ibm-amqQueueManagerName	The name of the queue manager or queue manager group to which an IBM MQ client application can request connection.	QMNAME
ibm-amqSecurityExitUserData	The user data that is passed to the security exit.	SCYDATA
ibm-amqSecurityExitName	The name of the exit program to be run by the channel security exit.	SCYEXIT
ibm-amqSslCipherSpec	A single CipherSpec for a TLS connection.	SSLCIPH
ibm-amqSslPeerName	Checks the Distinguished Name (DN) of the certificate from the peer queue manager or client at the other end of an IBM MQ channel.	SSLPEER
ibm-amqTransactionProgramName	The transaction program name.	TPNAME

Table 166. IBM MQ client channel directory string attributes (continued)		
LDAP Attribute	**Description**	**IBM MQ Property**
ibm-amqUserID	The user ID to be used by the MCA when attempting to initiate a secure SNA session with a remote MCA.	USERID

IBM MQ client connection integer attributes

The attributes with predefined values (for example, an enumerated type) are stored as standard integers. These values are stored in the LDAP directory as integer values, and not by using the associated constant name.

Table 167. IBM MQ client channel directory integer attributes		
LDAP attribute	**Description**	**IBM MQ Property**
ibm-amqConnectionAffinity	Determines whether client applications, which connect multiple times through the same queue manager name, use the same client channel.	AFFINITY
ibm-amqClientChannelWeight	A weighting to influence which client connection channel definition is used.	CLNTWGHT
ibm-amqHeartBeatInterval	The approximate time between heartbeat flows that are to be passed from a sending MCA when there are no messages on the transmission queue.	HBINT
ibm-amqKeepAliveInterval	A time-out value for a channel.	KAINT
ibm-amqMaximumMessageLength	The maximum length of a message that can be transmitted on the channel.	MAXMSGL
ibm-amqSharingConversations	The maximum number of conversations that share each TCP/IP channel instance.	SHARECNV
ibm-amqTransportType	The transport type to be used.	TRPTYPE

IBM MQ client channel boolean attribute

This Boolean attribute is not mapped to any IBM MQ property. The syntax of this attribute indicates a boolean value.

Table 168. IBM MQ client channel boolean attribute	
LDAP attribute	**Description**
ibm-amqIsClientDefault	This boolean attribute is defined to resolve the problem of searching entries whose ibm-amqQueueManagerName attribute has not been defined.

IBM MQ client channel list attributes

IBM MQ properties are stored as single-value, comma-separated list attribute within the LDAP directory. The attributes are defined in the same manner as the other directory string attributes. The list attributes along with their mapping to the IBM MQ properties are described in the following table.

Table 169. IBM MQ client channel list attributes		
LDAP attribute	**Description**	**IBM MQ Property**
ibm-amqHeaderCompression	A list of header data compression techniques supported by the channel.	COMPHDR
ibm-amqMessageCompression	A list of message data compression techniques supported by the channel.	COMPMSG
ibm-amqSendExitUserData	The user data that is passed to the send exit.	SENDDATA
ibm-amqSendExitUserName	The name of the exit program to be run by the channel send exit.	SENDEXIT
ibm-amqReceiveExitUserData	The user data that is passed to the receive exit.	RCVDATA
ibm-amqReceiveExitName	The name of the user exit program to be run by the channel receive user exit.	RCVEXIT

ALW *Common Name*

The common name (CN) consists of the channel name and the defining queue manager name.

It is a preexisting attribute.

The format of the CN is:

```
CN=CHANNEL_NAME(DEFINING_Q_MGR_NAME)
```

For example:

```
CN=TC1(QM_T1)
```

You can specify only one value for this attribute.

This attribute is a string attribute and the values are not case-sensitive. The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter, using a substring (for example, CN=jim* where CN is an attribute) and contains one or more wildcards.

ALW *ibm-amqChannelName*

This attribute specifies the name of the channel definition.

This attribute has a single string value with a maximum of 20 characters that are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter, using a substring and contains one or more wildcards.

ALW *ibm-amqDescription*

This LDAP attribute provides the channel description.

This attribute has a single string value with a maximum of 64 bytes, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqConnectionName*
This LDAP attribute is the communications connection identifier. It specifies the particular communications links to be used by this channel.

This attribute has a single string value with a maximum of 264 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqLocalAddress*
This attribute specifies the local communications address for the channel.

This attribute has a single string value with a maximum of 48 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqModeName*
This attribute is for use with LU 6.2 connections. It gives extra definition for the session characteristics of the connection when a communication session allocation is performed.

This attribute has a single string value of exactly 8 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqPassword*
This LDAP attribute specifies a password that can be used by the MCA when attempting to initiate a secure LU 6.2 session with a remote MCA.

This attribute has a single integer value with a maximum of 12 digits. It is not a pre-existing attribute.

ALW *ibm-amqQueueManagerName*
This attribute specifies the name of the queue manager or queue manager group to which an IBM MQ client application can request connection.

This attribute has a single string value with a maximum of 48 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

Related reference
“ibm-amqIsClientDefault” on page 1116
This Boolean attribute solves the problem of searching entries where the ibm-amqQueueManagerName attribute has not been defined.

ALW *ibm-amqSecurityExitUserData*
This LDAP attribute specifies user data that is passed to the security exit.

This attribute has a single string value with a maximum of 999 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqSecurityExitName*
This LDAP attribute specifies the name of the exit program to be run by the channel security exit.

Leave blank if no channel security exit is in effect.

This attribute has a single string value with a maximum of 999 characters, which are not case-sensitive. This attribute is not a pre-exiting one.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqSslCipherSpec*
This LDAP attribute specifies a single CipherSpec for a TLS connection.

This attribute has a single string value with a maximum of 32 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqSslPeerName*
This LDAP attribute is used to check the Distinguished Name (DN) of the certificate from the peer queue manager or client at the other end of an IBM MQ channel.

This LDAP attribute has a single string value with a maximum of 1024 bytes, which are not case-sensitive. It is not a pre-existing one.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqTransactionProgramName*
This LDAP attribute specifies the transaction program name. It is for use with LU 6.2 connections.

This attribute has a single string value with a maximum of 64 characters, which are not case-sensitive. It is not a pre-existing one.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqUserID*
This LDAP attribute specifies the user ID to be used by the MCA when attempting to initiate a secure SNA session with a remote MCA.

This attribute has a single string value of exactly 12 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

ALW *ibm-amqConnectionAffinity*
This LDAP attribute specifies whether client applications, which connect multiple times using the same queue manager name, use the same client channel.

This attribute has a single integer value. It is not a pre-existing attribute.

ALW *ibm-amqClientChannelWeight*
This LDAP attribute specifies a weighting that influences which client connection channel definition is used.

The client channel weighting attribute is used to bias the selection of client channel definitions when more than one suitable definition is available.

This attribute has a single integer value. It is not a pre-existing attribute.

ALW *ibm-amqHeartBeatInterval*
This LDAP attribute specifies the approximate time between heartbeat flows that are to be passed from a sending MCA when there are no messages on the transmission queue.

This attribute has a single integer value. It is not a pre-existing attribute. The default value is 1. The default is set in the current MQSERVER environment variable operation.

ALW *ibm-amqKeepAliveInterval*
This LDAP attribute is used to specify a time-out value for a channel.

The value of this attribute is passed to the communications stack specifying the keepalive timing for the channel. You can use this to specify a different keepalive value for each channel.

This attribute has a single integer value. It is not a pre-existing attribute.

ALW *ibm-amqMaximumMessageLength*
This LDAP attribute specifies the maximum length of a message that can be transmitted on the channel.

The default value of this attribute is 104857600 as per the current MQSERVER environment variable operation. This attribute has a single integer value and it is not a pre-existing attribute.

ALW *ibm-amqSharingConversations*
This LDAP attribute specifies the maximum number of conversations that share each TCP/IP channel instance.

This attribute has a single integer value. This attribute is not a pre-existing attribute.

ALW *ibm-amqTransportType*
This LDAP attribute specifies the transport type to be used.

This attribute has a single integer value. It is not a pre-existing attribute.

ALW *ibm-amqIsClientDefault*
This Boolean attribute solves the problem of searching entries where the ibm-amqQueueManagerName attribute has not been defined.

Preconnect exit modules generally search the LDAP servers with the value of the ibm-amqQueueManagerName attribute as the search criteria. Such a query would return all entries where the ibm-amqQueueManagerName attribute value matches the name of the queue manager specified on the MQCONN/X call. However when using the client channel definition tables (CCDT), you can either set the queue manager name on a MQCONN/X call as blank or prefix the name with an asterisk (*). If the name of the queue manager is blank, the client connects to the default queue manager. If the name is prefixed with an asterisk (*) to the queue manager, then the client connects any queue manager.

Similarly, the ibm-amqQueueManagerName attribute in an entry can be left undefined. In this case, it is expected that the client using this endpoint information can connect to any queue manager. For example, an entry contains the following lines:

```
ibm-amqChannelName = "CHANNEL1"
ibm-amqConnectionName = myhost(1414)
```

In this example, the client attempts to connect to the specified queue manager running on `myhost`.

However in LDAP Servers, a search is not made on an attribute value that has not been defined. For example, if an entry contains the connection information except ibm-amqQueueManagerName, then the search results would not include this entry. To overcome this problem, you can set ibm-amqIsClientDefault. This is a Boolean attribute and is assumed to have a value of FALSE if not defined.

For entries where the ibm-amqQueueManagerName has not been defined and are expected to be part of the search, set ibm-amqIsClientDefault to TRUE. When a blank or asterisk (*) is specified as the queue manager name in a call to MQCONN/X, the preconnect exit searches the LDAP sever for all entries where ibm-amqIsClientDefault attribute value is set to TRUE.

Note: Do not set or define ibm-amqQueueManagerName attribute if ibm-amqIsClientDefault is set to TRUE.

Related reference

"ibm-amqQueueManagerName" on page 1114
This attribute specifies the name of the queue manager or queue manager group to which an IBM MQ client application can request connection.

ALW *ibm-amqHeaderCompression*

This LDAP attribute is a list of header data compression techniques supported by the channel.

The maximum size of this attribute is of 48 characters. It is not a pre-existing attribute.

You can specify only one value for this attribute.

This list attribute is specified as directory strings using a comma-separated format. For example, the value specified for **ibm-amqHeaderCompression** is `0` which is mapped to `NONE`. Any values that exceed the maximum allowed limit are ignored by the client. For example, ibm-amqHeaderCompression contains a maximum of 2 integers in the list.

ALW *ibm-amqMessageCompression*

This LDAP attribute is a list of message data compression techniques supported by the channel.

The maximum size of this attribute is of 48 characters. It is not a pre-existing attribute.

This attribute does not support multiple values.

V 9.4.0 This list attribute is specified as directory strings using a comma-separated format. For example, the value specified for this attribute is 1,2,4,16,32 which maps to the underlying compression sequence RLE, ZLIBFAST, ZLIBHIGH, LZ4FAST, and LZ4HIGH.

Any values that exceed the maximum allowed limit are ignored by the client. For example, ibm-amqMessageCompression contains a maximum of 16 integers in the list.

ALW *ibm-amqSendExitUserData*

This LDAP attribute specifies user data that is passed to the send exit.

This LDAP attribute has a single string value with a maximum of 999 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

Note: ibm-amqSendExitName and **ibm-amqSendExitUserData** need to be synchronized in pairs . The user data should be synchronized with the exit name. So if one is specified, the other also has to be symmetrically specified, even if it contains no data.

ALW *ibm-amqSendExitName*

This LDAP attribute specifies the name of the exit program to be run by the channel send exit.

This attribute has a single string value with a maximum of 999 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

Note: **ibm-amqSendExitName** and **ibm-amqSendExitUserData** must be synchronized in pairs. The user data must be synchronized with the exit name. So if one is specified, the other also must be symmetrically specified even if it contains no data.

ALW *ibm-amqReceiveExitUserData*
This LDAP attribute specifies user data that is passed to the receive exit.

You can run a sequence of receive exits. The string of user data for a series of exits is separated by a comma, spaces, or both.

This attribute has a single string value with a maximum of 999 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

Note: **ibm-amqReceiveExitName** and **ibm-amqReceiveExitUserData** must be synchronized in pairs. The user data must be synchronized with the exit name. So if one is specified, the other also must be symmetrically specified even if it contains no data.

ALW *ibm-amqReceiveExitName*
This LDAP attribute specifies the name of the user exit program to be run by the channel receive user exit.

This attribute is a list of names of programs that are to be run in succession. Leave blank, if no channel receive user exit is in effect.

This attribute has a single string value with a maximum of 999 characters, which are not case-sensitive. It is not a pre-existing attribute.

The substring matching is ignored. The substring matching is a matching rule used in sub-schema that specifies the behavior of the attribute in a search filter.

Note: **ibm-amqReceiveExitName** and **ibm-amqReceiveExitUserData** must be synchronized in pairs. The user data must be synchronized with the exit name. So if one is specified, the other must also be symmetrically specified, even if it contains no data.

z/OS Using the sample programs for z/OS

The sample procedural applications that are delivered with IBM MQ for z/OS demonstrate typical uses of the Message Queue Interface (MQI).

About this task

IBM MQ for z/OS also provides sample data-conversion exits, described in "Writing data-conversion exits" on page 947.

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 about how to design panel-driven interfaces for non-programmable 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.

Procedure

- Use the following links to find out more about the sample programs:
  - "Features demonstrated in the sample applications for z/OS" on page 1119
  - "Preparing and running sample applications for the batch environment on z/OS" on page 1126
  - "Preparing sample applications for the TSO environment on z/OS" on page 1128

- “Preparing the sample applications for the CICS environment on z/OS” on page 1130
- “Preparing the sample application for the IMS environment on z/OS” on page 1133
- “The Put samples on z/OS” on page 1134
- “The Get samples on z/OS” on page 1136
- “The Browse sample on z/OS” on page 1139
- “The Print Message sample on z/OS” on page 1141
- “The Queue Attributes sample on z/OS” on page 1144
- “The Mail Manager sample on z/OS” on page 1145
- “The Credit Check sample on z/OS” on page 1152
- “The Message Handler sample on z/OS” on page 1163
- “The Asynchronous Put sample on z/OS” on page 1166
- “The Batch Asynchronous Consumption sample on z/OS” on page 1167
- “The CICS Asynchronous Consumption and Publish/Subscribe sample on z/OS” on page 1169
- “The Publish/Subscribe sample on z/OS” on page 1172
- “The Set and Inquire message property sample on z/OS” on page 1174

Related tasks

“Using the sample programs on Multiplatforms” on page 1018
These sample procedural programs are delivered with the product. The samples are written in C and COBOL, and demonstrate typical uses of the Message Queue Interface (MQI).

z/OS *Features demonstrated in the sample applications for z/OS*

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.

z/OS *Put samples on z/OS*

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 172 on page 1126 for the batch application and Table 179 on page 1131 for the CICS application.

z/OS *Get samples on z/OS*

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 172 on page 1126 for the batch application and Table 179 on page 1131 for the CICS application.

z/OS *Browse sample on z/OS*

The Browse sample demonstrates how to use the Browse option to find 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 173 on page 1127 for the batch application.

z/OS *Print Message sample on z/OS*

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. It can, optionally, display all of the message properties associated with each message.

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
- MQCRTMH
- MQDLTMH
- MQINQMP

The program is delivered in the C language. The application runs in the batch environment. See Table 174 on page 1127 for the batch application.

z/OS *Queue Attributes sample on z/OS*

The Queue Attributes sample demonstrates how to inquire about and set the values of IBM 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 180 on page 1131 for the CICS application.

z/OS *Mail Manager sample on z/OS*

Considerations to note when using 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 IBM MQ for z/OS batch adapter and include some ISPF panels.

See Table 177 on page 1129 for the TSO application, and Table 181 on page 1132 for the CICS application.

z/OS *Credit Check sample on z/OS*

This information contains points to consider when using 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 182 on page 1132 for the CICS application, and Table 184 on page 1134 for the IMS application.

z/OS *The Message Handler sample on z/OS*

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. SeeTable 178 on page 1130for the TSO application.

z/OS *Distributed queuing exit samples on z/OS*

A table of source programs of Distributed queuing exit samples.

The names of the source programs of the distributed queuing exit samples are listed in the following table:

Table 170. 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
CSQ4BCX4	C	Source program	SCSQC37S

Note: The source programs are link-edited with CSQXSTUB.

z/OS *Data-conversion exit samples on z/OS*

A skeleton is provided for a data-conversion exit routine, and a sample is shipped with IBM MQ illustrating the MQXCNVC call.

The names of the source programs of the data-conversion exit samples are listed in the following table:

Table 171. 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 947 for more information.

z/OS *Publish/Subscribe samples on z/OS*

The Publish/Subscribe sample programs demonstrate the use of the publish and subscribe features in IBM MQ.

There are four C and two COBOL programming language sample programs demonstrating how to program to the IBM MQ Publish/Subscribe interface.

The applications use these MQI calls:

- MQCONN
- MQOPEN
- MQPUT
- MQSUB
- MQGET
- MQCLOSE
- MQDISC
- MQCRTMH
- MQDLTMH
- MQINQMP

The Public/Subscribe sample programs are delivered in the C and COBOL programming languages. The sample applications run in the batch environment. See Publish/Subscribe samples for the batch applications.

z/OS *Configuring a queue manager to accept client connections on z/OS*

Before you can run the sample applications, you must first create a queue manager. You can then configure the queue manager to securely accept incoming connection requests from applications that are running in client mode.

Before you begin

Ensure the queue manager already exists and has been started. Determine whether channel authentication records are already enabled by issuing the MQSC command:

```
DISPLAY QMGR CHLAUTH
```

Important: This task expects that channel authentication records are enabled. If this is a queue manager used by other users and applications, changing this setting will affect all other users and applications. If

your queue manager does not make use of channel authentication records then step 4 can be replaced with an alternate authentication method (for example a security exit) which sets the MCAUSER to the *non-privileged-user-id* you will obtain in step "1" on page 1124.

You must know which channel name your application expects to use so that the application can be permitted to use the channel. You must also know which objects, for example queues or topics, your application expects to use so that your application can be permitted to use them.

About this task

This task creates a non-privileged user ID to be used for a client application which connects to the queue manager. Access is granted for the client application only to be able to use the channel it needs and the queue it needs by use of this user ID.

Procedure

1. Obtain a user ID on the system your queue manager is running on.

   For this task this user ID must not be a privileged administrative user. This user ID is the authority under which the client connection will run on the queue manager.
2. Start a listener program.
   a) Ensure that your channel initiator is started. If not, start it by issuing the **START CHINIT** command.
   b) Start the listener program by issuing the following command:

   ```
   START LISTENER TRPTYPE(TCP) PORT(nnnn)
   ```

   where *nnnn* is your chosen port number.
3. If your application uses the SYSTEM.DEF.SVRCONN then this channel is already defined. If your application uses another channel, create it by issuing the MQSC command:

   ```
   DEFINE CHANNEL(' channel-name ') CHLTYPE(SVRCONN) TRPTYPE(TCP) +
   DESCR('Channel for use by sample programs')
   ```

   channel-name is the name of your channel.
4. Create a channel authentication rule allowing only the IP address of your client system to use the channel by issuing the MQSC command:

   ```
   SET CHLAUTH(' channel-name ') TYPE(ADDRESSMAP) ADDRESS(' client-machine-IP-address ') +
   MCAUSER(' non-privileged-user-id ')
   ```

   where

   channel-name is the name of your channel.
   client-machine-IP-address is the IP address of your client system. If your sample client application is running on the same machine as the queue manager then use an IP address of '127.0.0.1' if your application is going to connect using 'localhost'. If several different client machines are going to connect in, you can use a pattern or a range instead of a single IP address. See Generic IP addresses for details.
   non-privileged-user-id is the user ID you obtained in step "1" on page 1124
5. If your application uses the SYSTEM.DEFAULT.LOCAL.QUEUE, then this queue is already defined. If your application uses another queue, create it by issuing the MQSC command:

   ```
   DEFINE QLOCAL(' queue-name ') DESCR('Queue for use by sample programs')
   ```

   where *queue-name* is the name of your queue.
6. Grant access to connect to and inquire the queue manager:

a) Ensure that your channel initiator is started. If not, start the channel initiator by issuing the START CHINIT command.

b) Start a TCP listener, for example issue the following command:

```
START LISTENER TRPTYPE(TCP) PORT(nnnn)
```

where *nnnn* is your chosen port number.

7. If your application is a point-to-point application, that is it makes use of queues, grant access to allow inquiring and the putting and getting messages using your queue by the user ID to be used, by issuing the MQSC commands:

   Issue the RACF commands:

```
RDEFINE MQQUEUE qmgr-name.QUEUE. queue-name UACC(NONE)

PERMIT qmgr-name.QUEUE. queue-name CLASS(MQQUEUE) ID(non-privileged-user-id) ACCESS(UPDATE)
```

   where

   qmgr-name is the name of your queue manager

   queue-name is the name of your queue.

   non-privileged-user-id is the user ID you obtained in step "1" on page 1124

8. If your application is a publish/subscribe application, that is it makes use of topics, grant access to allow publishing and subscribing using your topic by the user ID to be used, by issuing the following RACF commands:

```
RDEFINE MQTOPIC qmgr-name.PUBLISH.SYSTEM.BASE.TOPIC UACC(NONE)

PERMIT qmgr-name.PUBLISH.SYSTEM.BASE.TOPIC CLASS(MQTOPIC) ID(non-privileged-user-id)
ACCESS(UPDATE)

RDEFINE MQTOPIC qmgr-name.SUBSCRIBE.SYSTEM.BASE.TOPIC UACC(NONE)

PERMIT qmgr-name.SUBSCRIBE.SYSTEM.BASE.TOPIC CLASS(MQTOPIC) ID(non-privileged-user-id)
ACCESS(UPDATE)
```

   where

   qmgr-name is the name of your queue manager

   non-privileged-user-id is the user ID you obtained in step "1" on page 1124

   This will give *non-privileged-user-id* access to any topic in the topic tree, alternatively, you can define a topic object using **DEFINE TOPIC** and grant accesses only to the part of the topic tree referenced by that topic object. For more information, see Controlling user access to topics.

What to do next

Your client application can now connect to the queue manager and put or get messages using the queue.

Related concepts

z/OS Authority to work with IBM MQ objects on z/OS

Related reference

SET CHLAUTH

DEFINE CHANNEL

DEFINE QLOCAL

SET AUTHREC

z/OS *Preparing and running sample applications for the batch environment on z/OS*

To prepare a sample application that runs in the batch environment, perform the same steps that you would when building any batch IBM MQ for z/OS application.

These steps are listed in "Building z/OS batch applications" on page 985.

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 172 on page 1126, Table 173 on page 1127, Table 174 on page 1127, and Table 175 on page 1127.

You must edit the run JCL supplied for the samples that you want to use (see Table 172 on page 1126, Table 173 on page 1127, Table 174 on page 1127, and Table 175 on page 1127).

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.

z/OS *Names of the sample batch applications on z/OS*

A summary of the programs that are supplied for sample batch applications.

The batch application programs are summarized in the following tables:

- Table 172 on page 1126 Put and Get samples
- Table 173 on page 1127 Browse sample
- Table 174 on page 1127 Print message sample
- Table 175 on page 1127 Publish/Subscribe samples
- Table 176 on page 1128 Other samples

Table 172. Batch Put and Get samples

Member name	For language	Description	Source file supplied in library	Executable file supplied in library
CSQ4BCJ1	C	Get source program	SCSQC37S	SCSQLOAD
CSQ4BCK1	C	Put source program	SCSQC37S	SCSQLOAD
CSQ4BCJR	C	Sample run JCL for CSQ4BCJ1 and CSQBCK1	SCSQPROC	None
CSQ4BVJ1	COBOL	Get source program	SCSQCOBS	SCSQLOAD

Table 172. Batch Put and Get samples (continued)

Member name	For language	Description	Source file supplied in library	Executable file supplied in library
CSQ4BVK1	COBOL	Put source program	SCSQCOBS	SCSQLOAD
CSQ4BVJR	COBOL	Sample run JCL for CSQBVJ1 and CSQBVK1	SCSQPROC	None

Table 173. Batch Browse sample

Member name	For language	Description	Source file supplied in library	Executable file supplied in library
CSQ4BVA1	COBOL	Source program	SCSQCOBS	SCSQLOAD
CSQ4BVAR	COBOL	Sample run JCL for CSQ4BVA1	SCSQPROC	None
CSQ4BAA1	Assembler	Source program	SCSQASMS	SCSQLOAD
CSQ4BAAR	Assembler	Sample run JCL for CSQ4BAA1	SCSQPROC	None
CSQ4BCA1	C	Source program	SCSQC37S	SCSQLOAD
CSQ4BCAR	C	Sample run JCL for CSQ4BCA1	SCSQPROC	None
CSQ4BPA1	PL/I	Source program	SCSQPLIS	SCSQLOAD
CSQ4BPAR	PL/I	Sample run JCL for CSQ4BPA1	SCSQPROC	None

Table 174. Batch Print Message sample (C language only)

Member name	Description	Source file supplied in library	Executable file supplied in library
CSQ4BCG1	Source program	SCSQC37S	SCSQLOAD
CSQ4BCGR	Sample run JCL for CSQ4BCG1	SCSQPROC	None
CSQ4BCL1	Browse source program	SCSQC37S	SCSQLOAD
CSQ4BCLR	Sample run JCL for CSQ4BCL1	SCSQPROC	None

Table 175. Publish/Subscribe samples

Member name	For language	Description	Source file supplied in library	JCL in SCSQPROC	Executable file supplied in library
CSQ4BCP1	C	Publish to topic source program	SCSQC37S	CSQ4BCPP	SCSQLOAD
CSQ4BCP2	C	Subscribe to topic and get messages source program	SCSQC37S	CSQ4BCPS	SCSQLOAD

Table 175. Publish/Subscribe samples (continued)					
Member name	**For language**	**Description**	**Source file supplied in library**	**JCL in SCSQPROC**	**Executable file supplied in library**
CSQ4BCP3	C	Subscribe to topic using a user provided destination and get messages source program	SCSQC37S	CSQ4BCPD	SCSQLOAD
CSQ4BCP4	C	Subscribe to topic using extended options and get messages source program	SCSQC37S	CSQ4BCPE	SCSQLOAD
CSQ4BVP1	COBOL	Publish to topic source program	SCSQCOBS	CSQ4BVPP	SCSQLOAD
CSQ4BVP2	COBOL	Subscribe to topic and get messages source program	SCSQCOBS	CSQ4BVPS	SCSQLOAD

Table 176. Other samples					
Member name	**For language**	**Description**	**Source file supplied in library**	**JCL in SCSQPROC**	**Executable file supplied in library**
CSQ4BCS1	C	Asynchronous consumption source program	SCSQC37S	CSQ4BCSC	SCSQLOAD
CSQ4BCS2	C	Asynchronous Put, and Check status source program	SCSQC37S	CSQ4BCSP	SCSQLOAD
CSQ4BCM1	C	Inquire message properties source program	SCSQC37S	CSQ4BCMP	SCSQLOAD
CSQ4BCM2	C	Set message properties source program	SCSQC37S	CSQ4BCMP	SCSQLOAD

z/OS *Preparing sample applications for the TSO environment on z/OS*

To prepare a sample application that runs in the TSO environment, perform the same steps that you would when building any batch IBM MQ for z/OS application.

These steps are listed in “Building z/OS batch applications” on page 985. The library members to use are listed in Table 177 on page 1129.

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.

z/OS *Names of the sample TSO applications on z/OS*

Information about the names of the programs that are supplied for each of the sample TSO applications, and the libraries where the source, JCL, and, for the Message Handler sample only, the executable files reside.

The TSO application programs are summarized in the following tables:

- Table 177 on page 1129 Mail manager sample
- Table 178 on page 1130 Message handler sample

These samples use ISPF panels. You must therefore include the ISPF stub, ISPLINK, when you link-edit the programs.

Table 177. TSO Mail Manager sample

Member name	For language	Description	Source file supplied in library
CSQ4CVD	independent	IBM 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 178. TSO Message Handler sample				
Member name	**For language**	**Description**	**Source file supplied in library**	**Executable file 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
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

z/OS *Preparing the sample applications for the CICS environment on z/OS*

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 IBM MQ for z/OS application. These steps are listed in "Building CICS applications in z/OS" on page 988. The library members to use are listed in Table 179 on page 1131, Table 180 on page 1131, Table 181 on page 1132, and Table 182 on page 1132.

   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 CICS-IBM MQ Adapter* section in the CICS Transaction Server for z/OS 4.1 product documentation at: CICS Transaction Server for z/OS 4.1, The CICS-IBM MQ adapter.

   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.

z/OS *Names of the sample CICS applications on z/OS*

This topic provides a summary of the programs supplied for sample CICS applications.

The CICS application programs are summarized in the following tables:

- Table 179 on page 1131 Put and Get samples
- Table 180 on page 1131 Queue Attributes sample
- Table 181 on page 1132 Mail Manager sample (COBOL only)
- Table 182 on page 1132 Credit Check sample
- Table 183 on page 1133 Asynchronous Consumption and Publish/Subscribe samples

Table 179. CICS Put and Get samples

Member name	For language	Description	Source file supplied in library	Executable file supplied in library
CSQ4CCK1	C	Put source program	SCSQC37S	SCSQCICS
CSQ4CCJ1	C	Get source program	SCSQC37S	SCSQCICS
CSQ4CVJ1	COBOL	Get source program	SCSQCOBS	SCSQCICS
CSQ4CVK1	COBOL	Put source program	SCSQCOBS	SCSQCICS
CSQ4S100	independent	CICS system definition data set	SCSQPROC	None

Table 180. CICS Queue Attributes sample

Member name	For language	Description	Source file supplied in library	Executable file 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
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)

Table 180. CICS Queue Attributes sample (continued)				
Member name	**For language**	**Description**	**Source file supplied in library**	**Executable file supplied in library**
CSQ4S100	independent	CICS system definition data set	SCSQPROC	None

Table 181. CICS Mail Manager sample (COBOL only)		
Member name	**Description**	**Source file supplied in library**
CSQ4CVD	IBM 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 182. CICS Credit Check sample			
Member name	**For language**	**Description**	**Source file supplied in library**
CSQ4CVB	independent	IBM MQ object definitions	SCSQPROC
CSQ4CCB	independent	IBM 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
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

Table 182. CICS Credit Check sample (continued)			
Member name	**For language**	**Description**	**Source file supplied in library**
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

Table 183. CICS Asynchronous Consumption and Publish/Subscribe samples		
Member name	**Description**	**Source file supplied in library**
CSQ4CVCN	Source for Simple Message Consumption program	SCSQCOBS
CSQ4CVCT	Source for Control Message Consumption program	SCSQCOBS
CSQ4CVEV	Source for Event Handler program	SCSQCOBS
CSQ4CVPT	Source for Message Put Client program	SCSQCOBS
CSQ4CVRG	Source for Registration Client program	SCSQCOBS
CSQ4S100	CICS System Definition data set	SCSQPROC

z/OS *Preparing the sample application for the IMS environment on z/OS*

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 z/OS” on page 1130.

Then perform the following steps:

1. Perform the same steps that you would when building any IMS IBM MQ for z/OS application. These steps are listed in “Building IMS (BMP or MPP) applications” on page 989. The library members to use are listed in Table 184 on page 1134.
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.

z/OS *Names of the sample IMS application on z/OS*

This information provides a table with the list of the sources and JCLs that are supplied for the Credit Check sample IMS application.

Table 184. Source and JCL for the Credit Check IMS sample (C only)

Member name	Description	Supplied in library
CSQ4CVB	IBM 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

z/OS ***The Put samples on z/OS***

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 172 on page 1126 and Table 179 on page 1131).

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 1136. 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 1137. 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 1136. 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.

z/OS *The Put samples for the batch environment on z/OS*

Use this topic when considering 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 z/OS” on page 1126.

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 these 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 in the range 1 through 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 1-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.

z/OS *The Put samples for the CICS environment on z/OS*

Use this topic when considering 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 these 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 in the range 1 through 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.

z/OS *The Get samples on z/OS*

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 172 on page 1126 and Table 179 on page 1131).

z/OS *Design of the Get sample on z/OS*

Learn about the design of the Get sample, and some usage notes to consider.

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" on page 1136. 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 MQGET. 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" on page 1137. 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 1136. 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.

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 in the range 1 through 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.

z/OS *The Get samples for the batch environment on z/OS*

To run the samples, edit and run the sample JCL, as described in "Preparing and running sample applications for the batch environment on z/OS" on page 1126.

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 these 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
```

z/OS *The Get samples for the CICS environment on z/OS*

Special considerations for 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 these 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 in the range 1 through 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.

z/OS *The Browse sample on z/OS*

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 173 on page 1127).

To start the application, edit and run the sample run JCL, as described in “Preparing and running sample applications for the batch environment on z/OS” on page 1126. 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.

z/OS *Design of the Browse sample on z/OS*

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 z/OS" on page 1140.

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" on page 1140. 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" on page 1139). 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" on page 1140 and "3" on page 1140
   - 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 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" on page 1140 ; 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 1140.
8. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step "2" on page 1140.
9. Close the print data set and stop processing.

z/OS *Language-dependent design considerations on z/OS*

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.

z/OS *The Print Message sample on z/OS*

The Print Message sample is a batch application that demonstrates how to remove all the messages from a queue using the MQGET call.

The Print Message sample uses three parameters:

1. The name of the queue manager
2. The name of the source queue
3. An optional parameter for properties

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.

Permissible values for the property parameter are:

Table 185. Permissible values for the property parameter

Value	Behavior
0	Default behavior. 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 quotation marks, and byte strings are surrounded with X and single quotation marks, as for the message descriptor.

2	MQGMO_NO_PROPERTIES is specified, so that only message descriptor properties will be returned.
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 an IBM MQ property is included, otherwise the properties are discarded.

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 Print Message sample on z/OS” on page 1142. 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 174 on page 1127).

To start the application, edit and run the sample run JCL, as described in "Preparing and running sample applications for the batch environment on z/OS" on page 1126. When you run the application (and there are some messages on the queue), the output data set looks like that in Figure 139 on page 1142.

```
CSQ4BCG1 - starts here
**********************

MQCONN to MQ1E
MQOPEN - 'TEST.QUEUE'
MQCRTMH


MQGET of message number 1
****Message descriptor****
StrucId : 'MD ' Version : 2
Report  : 0 MsgType : 8
Expiry  : -1 Feedback : 0
Encoding : 785 CodedCharSetId : 500
Format : 'MQSTR   '
Priority : 0 Persistence : 0
MsgId : X'C3E2D840D4D8F1C540404040404040C1EA537F03167D88'
CorrelId : X'C3E2D840D4D8F1C540404040404040C1EA537F0317A928'
BackoutCount : 0
ReplyToQ    : '                          '
ReplyToQMgr  : ''
** Identity Context
UserIdentifier : 'FRED    '
AccountingToken :
X'0000000000000000000000000000000000000000000000000000000000000000'
ApplIdentityData : '                '
** Origin Context
PutApplType  : '2'
PutApplName  : 'FRED6            '
PutDate : '20080207'  PutTime : '17373745'
ApplOriginData : '   '
GroupId : X'000000000000000000000000000000000000000000000000'
MsgSeqNumber  : '1'
Offset     : '0'
MsgFlags    : '0'
OriginalLength : '-1'

****Message properties****
None

****  Message   ****

length - 30 bytes

00000000: E388 89A2 4089 A240 8140 A289 9497 9385 'This is a simple'
00000010: 40A3 85A2 A340 9485 A2A2 8187 855A   ' test message! '



No more messages
MQDLTMH
MQCLOSE
MQDISC
```

Figure 139. Example of a report from the Print Message sample application

z/OS *Design of the Print Message sample on z/OS*

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 beginning 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" on page 1142. 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" on page 1142). If this call is not successful, print the completion and reason codes and stop processing; otherwise, print the name of the queue.

4. If you use a message handle to obtain the message properties use MQCRTMH to create such a handle for use with subsequent MQGET calls. If this call is not successful, print the completion and reason codes and stop processing.
5. Set the get message options to reflect the request action for any message properties.
6. 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" on page 1142 and "3" on page 1142.
      - 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 beginning 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 64KB 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. If you created a message handle in step "4" on page 1143 call the printProperties subroutine to display any message properties.
   f. 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)
7. 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 "9" on page 1144.

   Note: The MQGET call fails if it finds a message that has more than 64KB 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 64KB 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

8. If you created a message handle in step “4” on page 1143 call MQDLTMH to delete it.
9. Close the queue using the MQCLOSE call with the object handle returned in step “3” on page 1142.
10. Disconnect from the queue manager using the MQDISC call with the connection handle returned in step “2” on page 1142.

z/OS *The Queue Attributes sample on z/OS*

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 180 on page 1131).

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 about writing reenterable programs, see Coding in System/390 assembler language.

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.

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.

z/OS *The Mail Manager sample on z/OS*

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 IBM 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

z/OS *Preparing the Mail Manager sample on z/OS*

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 IBM MQ, described in Channel control function.

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 z/OS” on page 1128.
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.

z/OS *Running the Mail Manager sample on z/OS*

To start the sample in the CICS Transaction Server for z/OS 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 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 142 on page 1150. When you select a message from this list, the contents of the message are displayed (see Figure 143 on page 1151).

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 complete 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.

z/OS *Design of the Mail Manager sample on z/OS*

The following sections describe each of the programs that make up the Mail Manager sample application.

The relationships between the programs and the panels that the application uses is shown in Figure 140 on page 1148 for the TSO version, and Figure 141 on page 1149 for the CICS Transaction Server for z/OS version.

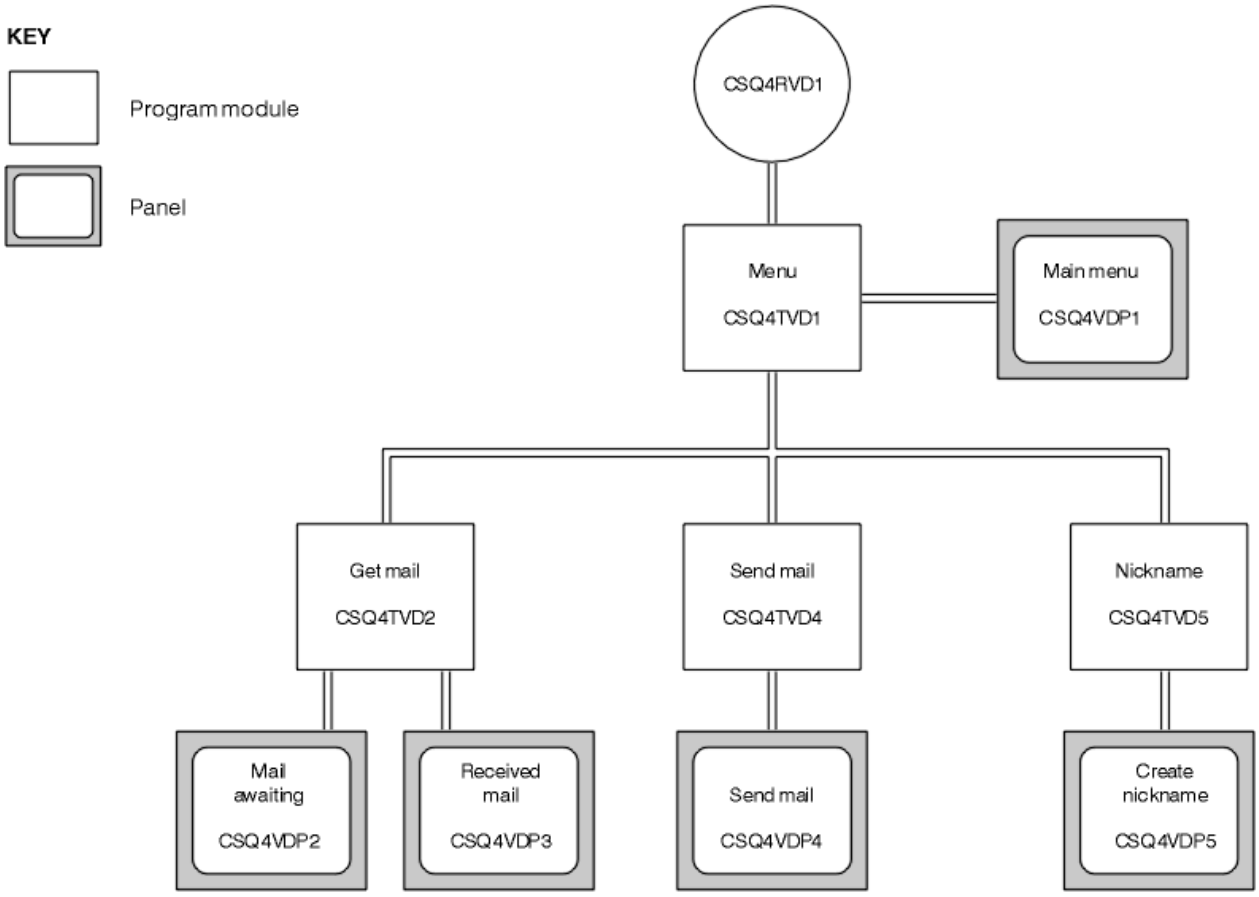


Figure 140. Programs and panels for the TSO versions of the Mail Manager

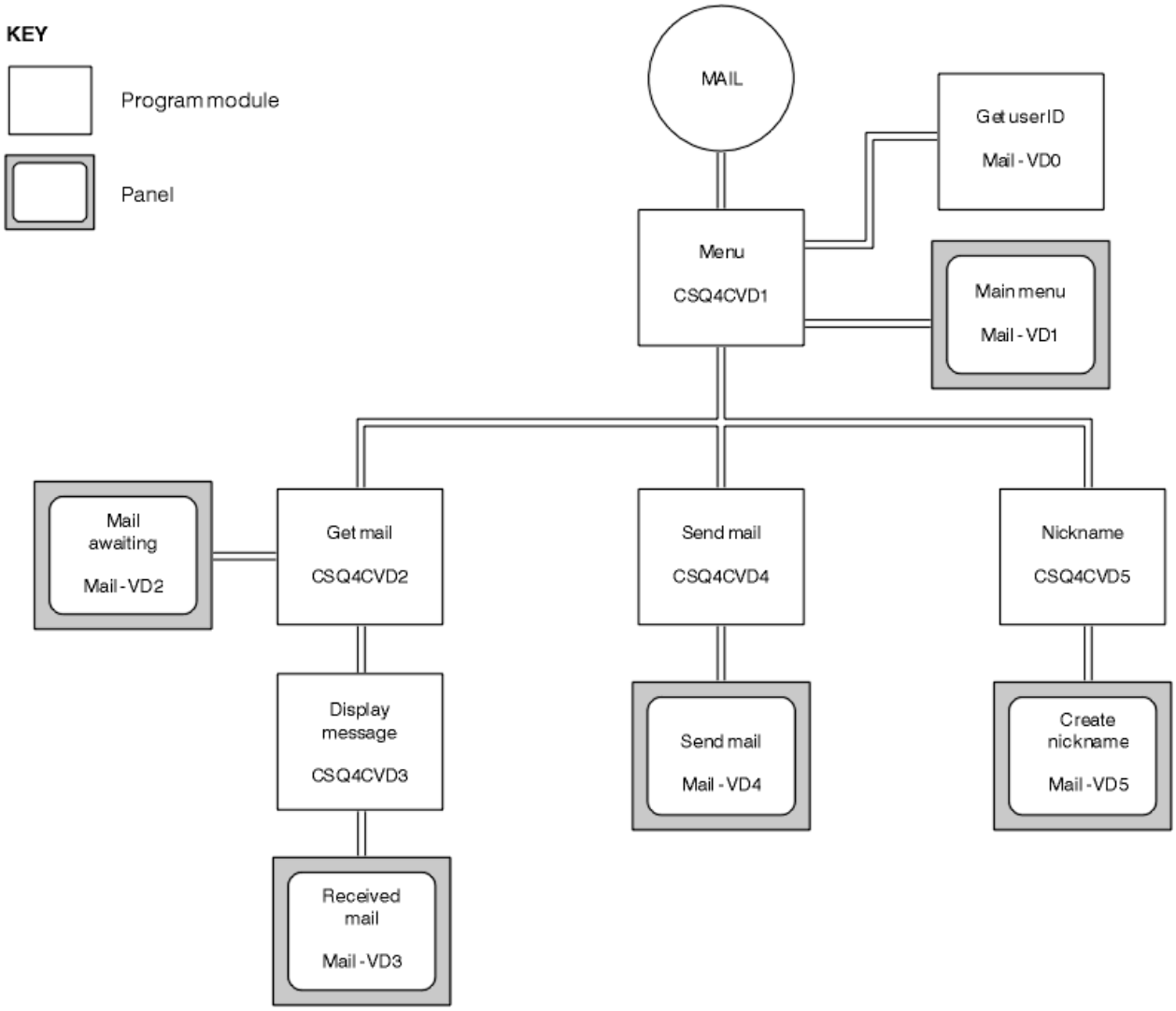


Figure 141. Programs and panels for the CICS version of the Mail Manager

z/OS *Menu program on z/OS*

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 sign on 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 IBM 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.

z/OS *Get-mail and display-message programs on z/OS*

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 142 on page 1150 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.

```
---------------------- IBM 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 142. 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 143 on page 1151 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.

```
--------------------- IBM MQ for z/OS Sample Programs ---------------------
COMMAND ==>
USERID - NTSFV02
Mail Manager System      QMGR  - VC4
Received Mail

Mail sent from JOHNJ  at VC4

Sent on the 01/06/1993 at 12:52:02
----------------------------------- Message ------------------------------
| HELLO FROM JOHNJ                                |
|                                     |
|                                     |
|                                     |
|                                     |
|                                     |
|                                     |
|                                     |
|                                     |
'-------------------------------------------------------------------------'
```

Figure 143. 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.

z/OS *Send-mail program on z/OS*

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.

z/OS *Nickname program on z/OS*

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 IBM 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.

z/OS *The Credit Check sample on z/OS*

The Credit Check sample application is a suite of programs that demonstrates how to use many of the features provided by IBM 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 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. This extension to the sample is described in "The IMS extension to the Credit Check sample on z/OS" on page 1162.

You can also run the sample on more than one queue manager, and send messages between each instance of the application. To do so, see "The Credit Check sample with multiple queue managers on z/OS" on page 1161.

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 182 on page 1132 and Table 184 on page 1134.

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.

z/OS *Preparing and running the Credit Check sample on z/OS*

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 z/OS" on page 1130. (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 z/OS" on page 1162.)
3. Start the CKTI trigger monitor (supplied with IBM 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 144 on page 1153 shows the Immediate Inquiry panel.

```
CSQ4VB2       IBM 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 144. 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" on page 1153 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.

z/OS *Design of the Credit Check sample on z/OS*

This section describes the design of each of the programs that make up the Credit Check sample application.

For more information about of some of the techniques that were considered during the design of the application, see "Design considerations for the Credit check sample on z/OS" on page 1159.

Figure 145 on page 1154 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.

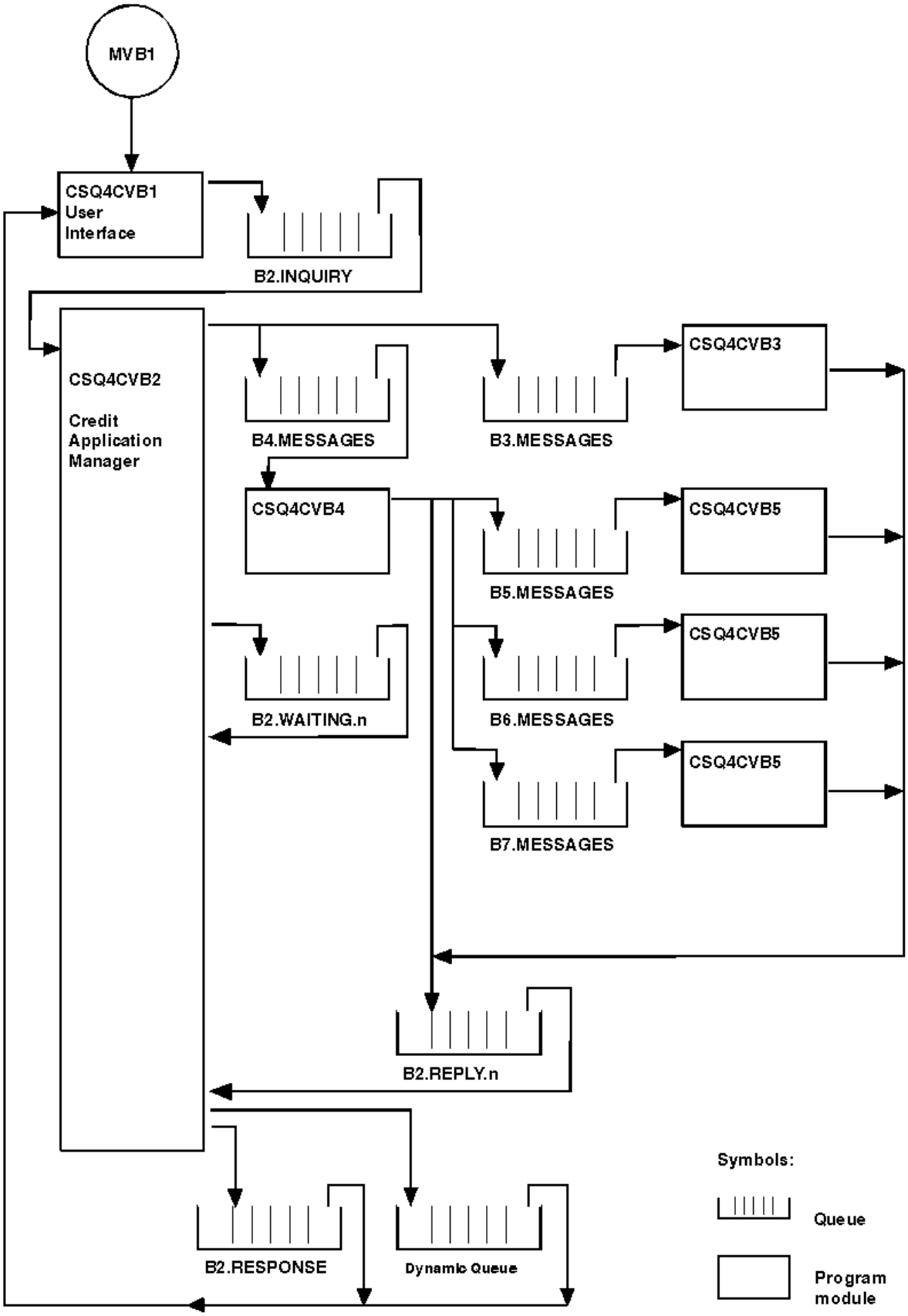


Figure 145. Programs and queues for the Credit Check sample application (COBOL programs only)

z/OS *User interface program (CSQ4CVB1) on z/OS*

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.

z/OS *Credit application manager (CSQ4CVB2) on z/OS*

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 IBM 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 1159.

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 from the enquiry queue

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 by the CAM

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 described in "Processing the message retrieved by the CAM on z/OS" on page 1156.

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.

z/OS *Processing the message retrieved by the CAM on z/OS*

A message retrieved by the Credit Application Manager (CAM) can be one of four types. The way in which the CAM processes a message depends on its type.

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 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 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 a unknown `CorrelIds`.

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 about the design of this part of the program, see "How the sample handles unexpected messages" on page 1160.

z/OS *Checking-account program (CSQ4CVB3) on z/OS*

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.

z/OS *Distribution program (CSQ4CVB4) on z/OS*

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 z/OS" on page 1161.

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.

z/OS *Agency-query program (CSQ4CVB5/CSQ4CCB5) on z/OS*

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 an IBM MQ application, and that the program modules that make up 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 IBM MQ for z/OS operations and control panels
- The ALTER NAMELIST command
- The CSQUTIL utility

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.)

z/OS *Design considerations for the Credit check sample on z/OS*

Design considerations for the Credit Check sample.

This topic contains information about:

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 is being used, this 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 is 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 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.
- A 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 effect 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.

z/OS *The Credit Check sample with multiple queue managers on z/OS*

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 Configuring distributed queuing.
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 DEFINE QREMOTE and DEFINE queues.

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 DEFINE NAMELIST.

z/OS *The IMS extension to the Credit Check sample on z/OS*

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 z/OS" on page 1152.
2. Follow the steps in "Preparing the sample application for the IMS environment on z/OS" on page 1133.
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 IBM MQ for z/OS operations and control panels
   - The ALTER QALIAS command .

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 IBM MQ for z/OS operations and control panels
- The ALTER NAMELIST command.

You can then run the sample from CICS transaction MVB1. 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 IBM 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 IBM 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 IBM 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.

z/OS *The Message Handler sample on z/OS*

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 z/OS" on page 1128.
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.

z/OS *Using the Message Handler sample on z/OS*

Having installed the sample and invoked it from the tailored CLIST CSQ4RCH1, the screen shown in Figure 146 on page 1163 is displayed.

```
----------------------- IBM 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 146. 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 147 on page 1164).

```
---------------------- IBM 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 147. 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 148 on page 1165.

```
---------------------- IBM 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     : `00000000000000000000000000000000000000000000000000`X
BackoutCount    : 000000000
ReplyToQ     : `QL.TEST.ISCRES1                `
ReplyToQMgr    : `VM03                      `
UserIdentifier  : `NTSFV02   `
AccountingToken :
`06F2F5F5F3F0F1000000000000000000000000000000000000000000000000`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 .....T......`
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 148. 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.

Note: When you use the SCSQCLST(CSQ4RCH1) sample to browse messages, each invocation causes the backout count of the message to increase. If you want to change the behavior of this sample, copy the sample and modify the contents as necessary. You should be aware that other applications that rely on this backout count can be influenced by this increasing count.

z/OS *Design of the sample Message Handler sample on z/OS*

This topic describes the design of each of the programs that make up 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 that 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 set correctly. 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.

z/OS *The Asynchronous Put sample on z/OS*

The Asynchronous Put sample program puts messages on a queue using the asynchronous MQPUT call. The sample also retrieves status information using the MQSTAT call.

The Asynchronous Put applications use these MQI calls:

- MQCONN
- MQOPEN

- MQPUT
- MQSTAT
- MQCLOSE
- MQDISC

The sample programs are delivered in the C programming language.

The Asynchronous Put applications run in the batch environment. See Other samples for the batch applications.

This topic also provides information about the design of the Asynchronous Consumption program, and running the CSQ4BCS2 sample.

- "Running the CSQ4BCS2 sample" on page 1167
- "Design of the Asynchronous Put sample program" on page 1167

Running the CSQ4BCS2 sample

This sample program takes up to six 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, CSQ4BCS2 connects to the default queue manager. Message content is provided through standard input (**SYSIN DD**).

There is a sample JCL to run the program, it resides in CSQ4BCSP.

Design of the Asynchronous Put sample program

The program uses the MQOPEN call with either the output options supplied, or with the MQOO_OUTPUT and MQOO_FAIL_IF_QUIESCING options, to open the target queue for putting messages.

If the program 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 subsequent MQI calls, default values are used 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_RESPONSE to create a datagram message containing the text of that line and asynchronously puts the message on the target queue. The program continues until it reaches the end of the input, or until 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 which returns 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.

Note: To observe what happens when an MQPUT error is detected by the MQSTAT call, set MAXDEPTH on the target queue to a low value.

z/OS *The Batch Asynchronous Consumption sample on z/OS*

The CSQ4BCS1 sample program is delivered in C, it demonstrates the use of MQCB and MQCTL to consume messages from multiple queues asynchronously.

The Asynchronous Consumption samples run in the batch environment. See Other samples for the batch applications.

There is also a COBOL sample which runs in the CICS environment, see "The CICS Asynchronous Consumption and Publish/Subscribe sample on z/OS" on page 1169.

The applications use these MQI calls:

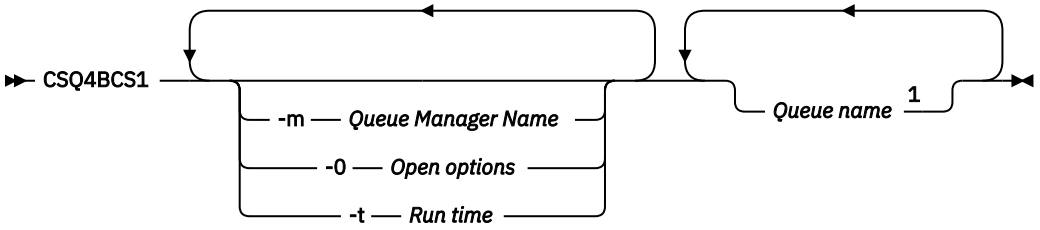
- MQCONN
- MQOPEN
- MQCLOSE
- MQDISC
- MQCB
- MQCTL

This topic also provided information about the following headings:

- "Running the CSQ4BCS1 sample" on page 1168
- "Design of the Batch Asynchronous Consumption sample program" on page 1168

Running the CSQ4BCS1 sample

This sample program follows the following syntax:



Notes:

[1] A maximum of 10 queues are supported by this sample. Provide more than one queue name in order to read messages from multiple queues.

There is a sample JCL to run this program, it resides in CSQ4BCSC.

Design of the Batch Asynchronous Consumption sample program

The sample shows how to read messages from multiple queues in the order of their arrival. This would require more code using synchronous MQGET. With asynchronous consumption, no polling is required, and thread and storage management is performed by IBM MQ. In the sample program, errors are written to the console.

The sample code has the following steps:

1. Define the single message consumption callback function.

```
void MessageConsumer(MQHCONN   hConn,
MQMD  * pMsgDesc,
MQGMO  * pGetMsgOpts,
MQBYTE * Buffer,
MQCBC  * pContext)
{ ... }
```

2. Connect to the queue manager.

```
MQCONN(QMName,&Hcon,&CompCode,&CReason);
```

3. Open the input queues, and associate each queue with the `MessageConsumer` callback function.

```
MQOPEN(Hcon,&od,O_options,&Hobj,&OpenCode,&Reason);
cbd.CallbackFunction = MessageConsumer;
MQCB(Hcon,MQOP_REGISTER,&cbd,Hobj,&md,&gmo,&CompCode,&Reason);
```

   `cbd.CallbackFunction` does not need to be set for each queue; it is an input-only field. You can associate a different callback function with each queue.

4. Start consumption of the messages.

```
MQCTL(Hcon,MQOP_START,&ctlo,&CompCode,&Reason);
```

5. Wait for the user to press Enter, then stop consumption of messages.

```
MQCTL(Hcon,MQOP_STOP,&ctlo,&CompCode,&Reason);
```

6. Finally, disconnect from the queue manager.

```
MQDISC(&Hcon,&CompCode,&Reason);
```

z/OS *The CICS Asynchronous Consumption and Publish/Subscribe sample on z/OS*

The Asynchronous Consumption and Publish/Subscribe sample programs demonstrate the use of asynchronous consumption, and publish and subscribe features within CICS.

A *Registration client* program registers three Callback handlers (an event handler, and two message consumers), and starts Asynchronous Consumption. A *Messaging client* program puts messages to a queue, or publishes suitable messages from a CICS console for consumption by the two Message Consumers (CSQ4CVCN and CSQ4CVCT).

To provide runtime control over the behavior of the sample, one of the message consumers can be instructed using the messages it receives, to SUSPEND, RESUME, or DEREGISTER any of the Callback handlers. It can also be used to issue an MQCTL STOP to end Asynchronous Consumption under control. The other message consumer is registered to subscribe to a topic.

Each program issues COBOL DISPLAY statements at appropriate points to display the behavior of the sample.

The applications use these MQI calls:

- MQOPEN
- MQPUT
- MQSUB
- MQGET
- MQCLOSE
- MQCB
- MQCTL

The programs are delivered in the COBOL language. See CICS Asynchronous Consumption and Publish/Subscribe samples for the CICS applications.

This topic also provides the following information:

- "Setup" on page 1170
- "Registration Client CSQ4CVRG" on page 1170
- "Event handler CSQ4CVEV" on page 1170
- "Simple Message Consumer CSQ4CVCN" on page 1170

- “Control Message Consumer CSQ4CVCT” on page 1170
- “Messaging Client CSQ4CVPT” on page 1170

Setup

The names of the Queue and Topic used by the Message Consumers are hardcoded in the Registration and Messaging Client programs.

The Queue, **SAMPLE.CONTROL.QUEUE**, should be defined to the Queue Manager associated with the CICS region before running the sample. The Topic, **News/Media/Movies**, can be defined if required, or it is created at runtime under the default Administrative Object if it does not exist.

CICS programs and transaction definitions can be installed by installing a group: CSQ4SAMP.

Registration Client CSQ4CVRG

The Registration Client program must be started under the CICS transaction MVRG. It takes no input.

When started, the Registration Client registers the following Callback handlers using MQCB:

- CSQ4CVEV as an Event Handler.
- CSQ4CVCN as a Message Consumer on a topic, **News/Media/Movies**.
- CSQ4CVCT as a Message Consumer on a Queue, **SAMPLE.CONTROL.QUEUE**.

The Registration Client passes a data structure containing the names of all three registered Callback handlers to CSQ4CVCT, together with the object handles associated with the two message consumers.

Having registered the Callback handlers, the Registration Client issues an MQCTL START_WAIT to start Asynchronous Consumption, and suspend until control is returned to it (for example, by one of the Callback handlers issuing an MQCTL STOP).

Event handler CSQ4CVEV

When driven, the Event Handler displays a message indicating the call type (for example, START). When driven for IBM MQ reason code CONNECTION_QUIESCING, the Event Handler issues an MQCTL STOP to end Asynchronous Consumption and return control to the Registration Client.

Simple Message Consumer CSQ4CVCN

When driven, this Message Consumer displays a message indicating the call type (for example, REGISTER). When driven for the MSG_REMOVED call type, the Message Consumer retrieves the inbound message and outputs it to the CICS job log.

Control Message Consumer CSQ4CVCT

When driven, this Message Consumer displays a message indicating the call type (for example, START). When driven for the MSG_REMOVED call type, the Message Consumer retrieves the inbound message and the data structure passed by the Registration Client. Based on the message content, it issues appropriate MQCB or MQCTL commands to one of the following:

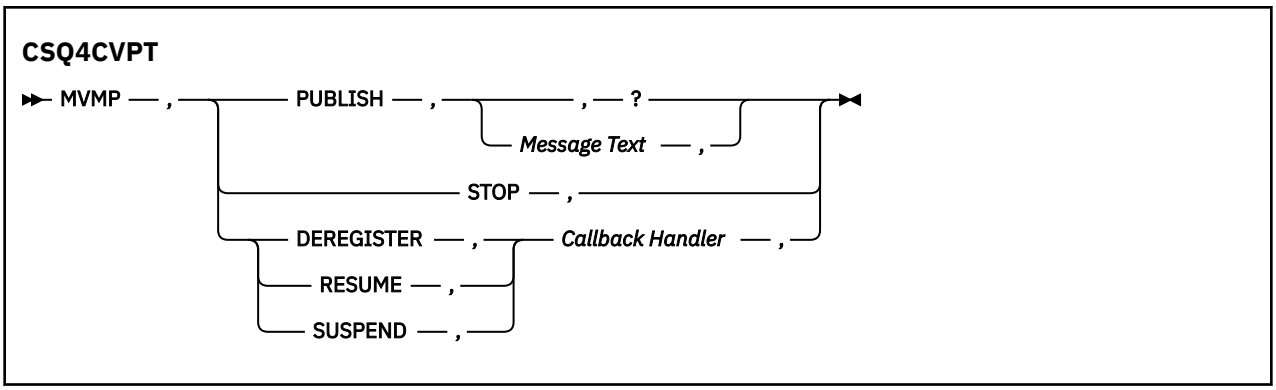
- STOP Asynchronous Consumption (returning control to the Registration Client).
- SUSPEND, RESUME, or DEREGISTER a named Callback handler (including itself).

Messaging Client CSQ4CVPT

The Messaging Client has two functions:

- It publishes a message to a topic for consumption by the Message Consumer CSQ4CVCN.
- It puts a control message to a queue for consumption by the Control Message Consumer CSQ4CVCT, resulting in a potential change in behavior of the sample.

The Messaging Client program must be started from a CICS console under a CICS transaction, and it takes command line input with the following syntax:



PUBLISH
Publish the Message Text (or a default message) as a Retained Message for consumption by the Simple Message Consumer.

STOP
Stop Asynchronous Consumption.

DEREGISTER
Deregister the named Callback handler.

RESUME
Resume the named Callback handler.

SUSPEND
Suspend the named Callback handler.

Input fields are positional, and comma-separated. Keywords and Callback Handler names are not case-sensitive.

Examples:

Table 186. Input examples

Example	Description
MVMP,PUBLISH,,	Publish a default message
MVMP,publish, `A short message`,	Publish the given text
MVMP,STOP,	Stop Asynchronous Consumption
MVMP,DEREGISTER,CSQ4CVEV,	Deregister the Event Handler
MVMP,resume,csq4cvcn,	Resume the Simple Message Consumer
MVMP,SUSPEND,CSQ4CVEV,	Suspend the Event Handler

Where MVMP is the CICS transaction associated with the Messaging Client program CSQ4CVPT.

Note:

- Suspending or deregistering all Callback handlers terminates the START_WAIT issued by the Registration Client, returning control to it, and ending the task.
- Suspending or deregistering the Control Callback Handler has deliberately not been prevented, but it removes the ability to further control the behavior of the sample.

z/OS *The Publish/Subscribe sample on z/OS*

The Publish/Subscribe sample programs demonstrate the use of the publish and subscribe features in IBM MQ.

There are four C and two COBOL programming language sample programs demonstrating how to program to the IBM MQ Publish/Subscribe interface. The programs are delivered in the C and COBOL language. The applications run in the batch environment; see Publish/Subscribe samples for the batch applications.

There are also COBOL samples that run in the CICS environment; see "The CICS Asynchronous Consumption and Publish/Subscribe sample on z/OS" on page 1169.

This topic also provides information about how to run Publish/Subscribe sample programs. These sample programs include:

Running the CSQ4BCP1 sample

This program is written in C; it publishes messages to a topic. Start one of the subscriber samples before running this program.

This program takes up to four parameters:

1. The name of the target topic string (required).
2. The name of the queue manager (optional).
3. Open options (optional).
4. Close options (optional).

If a queue manager is not specified, CSQ4BCP1 connects to the default queue manager. There is a sample JCL to run the program, it resides in CSQ4BCPP.

Message content is provided through standard input (**SYSIN DD**).

Running the CSQ4BCP2 sample

This program is written in C; it subscribes to a topic and prints the messages received.

This program takes up to three parameters:

1. The name of the target topic string (required).
2. The name of the queue manager (optional).
3. MQSD subscription options (optional).

If a queue manager is not specified, CSQ4BCP2 connects to the default queue manager. There is a sample JCL to run the program, it resides in CSQ4BCPS.

Running the CSQ4BCP3 sample

This program is written in C; it subscribes to a topic using a user-specified destination queue and prints the messages received.

This program takes up to four parameters:

1. The name of the target topic string (required).
2. The name of the destination (required).

3. The name of the queue manager (optional).
4. MQSD subscription options (optional).

If a queue manager is not specified, CSQ4BCP3 connects to the default queue manager. There is a sample JCL to run the program, it resides in CSQ4BCPD.

Running the CSQ4BCP4 sample

This program is written in C; it subscribes and gets messages from a topic allowing the use of extended options on the MQSUB call, extending those available on the simpler MQSUB sample: CSQ4BCP2. In addition to the message payload, message properties for each message are received and displayed.

This program takes a variable set of parameters:

- **-t** *Topic string*.
- **-o** *Topic object name*.

  Important: One of **-t** or **-o**, or both, is required
- **-m** *Queue manager name* (optional).
- **-b** *Connection binding type* (optional), where *type* can have any of the following values:
  - *standard* : MQCNO_STANDARD_BINDING , which is the default value
  - *shared*: MQCNO_SHARED_BINDING
  - *fastpath*: MQCNO_FASTPATH_BINDING
  - *isolated*: MQCNO_ISOLATED_BINDING
- **-q** *Destination queue name* (optional).
- **-w** *Wait interval on MQGET in seconds* (optional), where *seconds* can have any of the following values:
  - unlimited: MQWI_UNLIMITED
  - none: No wait
  - *n*: Wait interval in seconds
  - No value specified: When no value is specified, the default is 30 seconds
- **-d** *Subscription name* (optional). Creates or resumes named durable subscription.
- **-k** (optional). Keeps durable subscription on MQCLOSE.

If a queue manager is not specified, CSQ4BCP4 connects to the default queue manager. There is a sample JCL to run the program, it resides in CSQ4BCPE.

Running the CSQ4BVP1 sample

This program is written in COBOL, it publishes messages to a topic. Start one of the subscriber samples before running this program.

This program takes no parameters. **SYSIN DD** provides the input topic name, queue manager name, and message content.

If a queue manager is not specified, CSQ4BVP1 connects to the default queue manager. There is a sample JCL to run the program, it resides in CSQ4BVPP.

Running the CSQ4BVP2 sample

This program is written in COBOL, it subscribes to a topic and prints the messages received.

This program takes no parameters. **SYSIN DD** provides the input for topic name and queue manager name.

If a queue manager is not specified, CSQ4BVP1 connects to the default queue manager. There is a sample JCL to run the program, it resides in CSQ4BVPP.

z/OS *The Set and Inquire message property sample on z/OS*

The message property sample programs demonstrate the addition of user-defined properties to a message handle, and the inquisition of the properties associated with that message.

The applications use these MQI calls:

- MQCONN
- MQOPEN
- MQPUT
- MQGET
- MQCLOSE
- MQDISC
- MQCRTMH
- MQDLTMH
- MQINQMP
- MQSETMP

The programs are delivered in the C language. The applications run in the batch environment. See Other samples for the batch applications.

The CSQ4BCM1 program is used to inquire the properties of a message handle from a message queue, and it is an example of the use of the MQINQMP API call. The sample gets one message from a queue and then prints all the message handle properties.

The CSQ4BCM2 program is used to set the properties of a message handle on a message queue, and it is an example of the use of the MQSETMP API call. The sample creates a message handle and puts it into the `MsgHandle` field of the MQGMO structure. It then puts the message to a queue.

Other examples of inquiring and printing message properties are included in the CSQ4BCG1 and CSQ4BCP4 sample programs.

This topic also provides information on running the Set and Inquire message property samples under the following headings:

- "Running the CSQ4BCM1 sample" on page 1174
- "Running the CSQ4BCM2 sample" on page 1174

Running the CSQ4BCM1 sample

This program takes up to four 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).

Running the CSQ4BCM2 sample

This program takes up to six 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).

The property names, values, and message content are provided through the standard input (**SYSIN DD**). There is a sample JCL to run the program, it resides in CSQ4BCMP.

Developing applications for Managed File Transfer

Specify programs to run with Managed File Transfer, use Apache Ant with Managed File Transfer, customise Managed File Transfer with user exits, and control Managed File Transfer by putting messages on the agent command queue.

Specifying programs to run with MFT

You can run programs on a system where a Managed File Transfer Agent is running. As part of a file transfer request, you can specify a program to run either before a transfer starts, or after it finishes. Additionally, you can start a program that is not part of a file transfer request by submitting a managed call request.

About this task

There are five scenarios in which you can specify a program to run:

- As part of a transfer request, at the source agent, before the transfer starts.
- As part of a transfer request, at the destination agent, before the transfer starts.
- As part of a transfer request, at the source agent, after the transfer completes.
- As part of a transfer request, at the destination agent, after the transfer completes.
- Not as part of a transfer request. You can submit a request to an agent to run a program. This scenario is sometimes referred to as a managed call.

User exits and program calls are invoked in the following order:

```
- SourceTransferStartExit(onSourceTransferStart).
- PRE_SOURCE Command.
- DestinationTransferStartExits(onDestinationTransferStart).
- PRE_DESTINATION Command.
- The Transfer request is performed.
- DestinationTransferEndExits(onDestinationTransferENd).
- POST_DESTINATION Command.
- SourceTransferEndExits(onSourceTransferEnd.
- POST_SOURCE Command.
```

Notes:

1. The **DestinationTransferEndExits** is run only when the transfer completes, either successfully or partially successfully.
2. The **postDestinationCall** is run only when the transfer completes, either successfully or partially successfully.
3. The **SourceTransferEndExits** is run for successful, partially successful, or failed transfers.
4. The **postSourceCall** is called only if:
   - The transfer was not canceled.
   - There is a successful or partially successful outcome.
   - Any post-destination transfer programs ran successfully.

Procedure

- Specify the program that you want to run by using one of the following options:

Use an Apache Ant task

Use one of the `fte:filecopy`, `fte:filemove`, and `fte:call` Ant tasks to start a program. Using an Ant task, you can specify a program in any of the five scenarios, using the `fte:presrc`, `fte:predst`, `fte:postdst`, `fte:postsrc`, and `fte:command` nested elements. For more information, see Program invocation nested elements.

Edit the file transfer request message

You can edit the XML that is generated by a transfer request. Using this method, you can run a program in any of the five scenarios, by adding **preSourceCall**, **postSourceCall**, **preDestinationCall**, **postDestinationCall**, and **managedCall** elements to the XML file. Then, use this modified XML file as the transfer definition for a new file transfer request, for example with the **fteCreateTransfer -td** parameter. For more information, see MFT agent call request message examples.

Use the `fteCreateTransfer` command

You can use the **fteCreateTransfer** command to specify programs to start. You can use the command to specify programs to run in the first four scenarios, as part of a transfer request, but you cannot start a managed call. For information about the parameters to use, see **fteCreateTransfer**: start a new file transfer. For examples of using this command, see Examples of using fteCreateTransfer to start programs.

Related reference

commandPath MFT property

Managed calls

Managed File Transfer (MFT) agents are typically used to transfer files or messages. These are known as *Managed Transfers*. Agents can also be used to run commands, scripts or JCL without the need for transferring files or messages. This capability is known as *Managed Calls*.

Managed call requests can be submitted to an agent in several ways:

- Using the fte:call Ant task.
- Configuring a resource monitor with a task XML that runs a command or a script. See Configuring monitor tasks to start commands and scripts for more information.
- Directly putting an XML message into the agent's command queue. See File transfer request message format for more details on the Managed Call XML schema.

For managed calls, the directory containing the command or script that is being run must be specified in the agent property **commandPath**.

Managed calls cannot run commands or scripts that are located in directories which are not specified in the agent's **commandPath**. This is to ensure that the agent does not run any malicious code.

Important: To ensure this is the case, by default, when you specify **commandPath**:

- Any existing agent sandbox is configured by the agent when it starts up so that all **commandPath** directories are automatically added to the list of directories that have denied access for a transfer.
- Any existing user sandboxes are updated when the agent starts up so that all the **commandPath** directories (and their sub-directories) are added as `<exclude>` elements to the `<read>` and `<write>` elements.
- If the agent is not configured to use either an agent sandbox, or user sandboxes, then a new agent sandbox is created when the agent starts up that has the **commandPath** directories specified as denied directories.

Additionally, you can also enable authority checking on an agent to ensure that only authorized users are allowed to submit managed call requests. For more information on this, see Restricting user authorities on MFT agent actions.

The command, script, or JCL invoked as part of a managed call runs as an external process, which is monitored by the agent. When the process exits, the managed call completes and the return code from the process is made available to either the agent or the Ant script that invoked the **fte:call** Ant task.

If the managed call was started by the **fte:call** Ant task, then your Ant script can check the value of the return code to determine whether the managed call was successful or not.

For all other types of managed calls, you can specify which return code values should be used to indicate that the managed call completed successfully. The agent compares the return code from the process against these return codes when the external process finishes.

Note: Because managed calls run as external processes, they cannot be canceled once they have started.

Managed calls and source transfer slots

An agent contains a number of source transfer slots, as specified by the agent property **maxSourceTransfers**, described in Advanced agent properties: Transfer limit.

Whenever a managed call or a managed transfer is run, they occupy a source transfer slot. The slot is released when the managed call or managed transfer completes.

If all the source transfer slots are in use when an agent receives either a new managed call or managed transfer request, the request is queued by the agent until a slot becomes available.

If a managed call starts a managed transfer (for example, if a managed call runs an Ant script and that Ant script uses the fte:filecopy or fte:filemove task to transfer a file), then two source transfer slots are required:

- One for the managed transfer
- One for the managed call

In this situation, it is important to note that if the managed transfer either takes a long time to complete, or goes into recovery, then the two source transfer slots are occupied until either the managed transfer completes, is canceled or times out due to a **transferRecoveryTimeout**. See Transfer recovery timeout concepts for details on **transferRecoveryTimeout**. This can potentially limit the number of other managed transfers or managed calls that the agent can process.

Because of this, you should consider the design of a managed call to ensure that it does not occupy source transfer slots for a long period of time.

Using the REST API with managed calls

The HTTP GET and HTTP POST verbs are supported for enabling managed calls, and work only on Version 3 of the REST API.

Other verbs, for example, HTTP DELETE and HTTP UPDATE are not supported and return the HTTP 405 error code if you attempt to use them.

Attention: Once submitted a managed call cannot be canceled using the REST API.

Using Apache Ant with MFT

Managed File Transfer provides tasks that you can use to integrate file transfer function into the Apache Ant tool.

You can use the **fteAnt** command to run Ant tasks in a Managed File Transfer environment that you have already configured. You can use file transfer Ant tasks from your Ant scripts to coordinate complex file transfer operations from an interpreted scripting language.

For more information about Apache Ant, see the Apache Ant project web page: https://ant.apache.org/

Related concepts

"Getting started using Ant scripts with MFT" on page 1178
Using Ant scripts with Managed File Transfer allows you to coordinate complex file transfer operations from an interpreted scripting language.

fteAnt: run Ant tasks in MFT

Related reference

"Sample Ant tasks for MFT" on page 1179
There are a number of sample Ant scripts provided with your installation of Managed File Transfer. These samples are located in the directory *MQ_INSTALLATION_PATH*/mqft/samples/fteant. Each sample script contains an `init` target, edit the properties set in the `init` target to run these scripts with your configuration.

Getting started using Ant scripts with MFT

Using Ant scripts with Managed File Transfer allows you to coordinate complex file transfer operations from an interpreted scripting language.

Ant scripts

Ant scripts (or build files) are XML documents defining one or more targets. These targets contain task elements to run. Managed File Transfer provides tasks which you can use to integrate file transfer function into Apache Ant. To learn about Ant scripts, see the Apache Ant project web page: https://ant.apache.org/

Examples of Ant scripts that use Managed File Transfer tasks are provided with your product installation in the directory *MQ_INSTALLATION_PATH*/mqft/samples/fteant

On protocol bridge agents, Ant scripts are run on the protocol bridge agent system. These Ant scripts do not have direct access to the files on the FTP or SFTP server.

Namespace

A namespace is used to differentiate the file transfer Ant tasks from other Ant tasks that might share the same name. You define the namespace in the project tag of your Ant script.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:fte="antlib:com.ibm.wmqfte.ant.taskdefs" default="do_ping">

  <target name="do_ping">
     <fte:ping cmdqm="qm@localhost@1414@SYSTEM.DEF.SVRCONN" agent="agent1@qm1"
      rcproperty="ping.rc" timeout="15"/>
  </target>

</project>
```

The attribute `xmlns:fte="antlib:com.ibm.wmqfte.ant.taskdefs"` tells Ant to look for the definitions of tasks prefixed by `fte` in the library `com.ibm.wmqfte.ant.taskdefs`.

You do not need to use `fte` as your namespace prefix; you can use any value. The namespace prefix `fte` is used in all examples and sample Ant scripts.

Running Ant scripts

To run Ant scripts that contain the file transfer Ant tasks use the **fteAnt** command. For example:

```
fteAnt -file ant_script_location/ant_script_name
```

For more information, see **fteAnt**: run Ant tasks in MFT.

Return codes

The file transfer Ant tasks return the same return codes as the Managed File Transfer commands. For more information, see Return codes for MFT.

Related reference

fteAnt: run Ant tasks in MFT

"Sample Ant tasks for MFT" on page 1179

There are a number of sample Ant scripts provided with your installation of Managed File Transfer. These samples are located in the directory *MQ_INSTALLATION_PATH*/mqft/samples/fteant. Each sample script contains an `init` target, edit the properties set in the `init` target to run these scripts with your configuration.

Sample Ant tasks for MFT

There are a number of sample Ant scripts provided with your installation of Managed File Transfer. These samples are located in the directory *MQ_INSTALLATION_PATH*/mqft/samples/fteant. Each sample script contains an `init` target, edit the properties set in the `init` target to run these scripts with your configuration.

email

The email sample demonstrates how to use Ant tasks to transfer a file and send an email to a specified email address if the transfer fails. The script checks that the source and destination agents are active and able to process transfers by using the Managed File Transfer ping task. If both agents are active, the script uses the Managed File Transfer fte:filecopy task to transfer a file between the source and destination agents, without deleting the original file. If the transfer fails the script sends an email containing information about the failure by using the standard Ant email task.

hub

The hub sample is made up of two scripts: `hubcopy.xml` and `hubprocess.xml`. The `hubcopy.xml` script shows how you can use Ant scripting to build 'hub and spoke' style topologies. In this sample, two files are transferred from agents running on spoke machines to an agent running on the hub machine. Both files are transferred at the same time, and when the transfers are complete the `hubprocess.xml` Ant script is run on the hub machine to process the files. If both files transfer correctly, the Ant script concatenates the contents of the files. If the files do not transfer correctly, the Ant script cleans up by deleting any file data that was transferred. For this example to work correctly, you must put the `hubprocess.xml` script on the command path of the hub agent. For more information about setting the command path of an agent, see commandPath MFT property.

librarytransfer (IBM i platform only)

IBM i

IBM i The librarytransfer sample demonstrates how to use Ant tasks to transfer an IBM i library on one IBM i system to a second IBM i system.

IBM i The librarytransfer sample uses the native save file support on IBM i with predefined Ant tasks available in Managed File Transfer to transfer native library objects between two IBM i systems. The sample uses a <presrc> nested element in a Managed File Transfer filecopy task to invoke an executable script `librarysave.sh` that saves the requested library on the source agent system into a temporary save file. The save file is moved by the filecopy ant task to the destination agent system where a <postdst> nested element is used to invoke the executable script `libraryrestore.sh` to restore the library saved in the save file to the destination system.

IBM i Before you run this sample, you need to complete some configuration as described in the `librarytransfer.xml` file. You must also have a working Managed File Transfer environment on two IBM i machines. The setup must consist of a source agent running on the first IBM i machine and a destination agent running on the second IBM i machine. The two agents must be able to communicate with each other.

IBM i The librarytransfer sample consists of the following three files:

- `librarytransfer.xml`
- `librarysave.sh` (<presrc> executable script)
- `libraryrestore.sh` (<postdst> executable script)

The sample files are located in the following directory: `/QIBM/ProdData/WMQFTE/V7/samples/fteant/ibmi/librarytransfer`

IBM i To run this sample the user must complete the following steps:

1. Start a Qshell session. At an IBM i command window type: STRQSH
2. Change directory to the `bin` directory as follows:

```
cd  /QIBM/ProdData/WMQFTE/V7/bin
```

3. After completing the required configuration, run the sample by using the following command:

```
fteant -f /QIBM/ProdData/WMQFTE/V7/samples/fteant/ibmi/librarytransfer/librarytransfer.xml
```

physicalfiletransfer (IBM i platform only)

IBM i The physicalfiletransfer sample demonstrates how to use Ant tasks to transfer a Source Physical or Database file from a library on one IBM i system to a library on a second IBM i system.

IBM i The physicalfiletransfer sample uses the native save file support on IBM i with predefined Ant Tasks available in Managed File Transfer to transfer complete Source Physical and Database files between two IBM i systems. The sample uses a <presrc> nested element within a Managed File Transfer filecopy task to invoke an executable script `physicalfilesave.sh` to save the requested Source Physical or Database file from a library on the source agent system into a temporary save file. The save file is moved by the filecopy ant task to the destination agent system where a <postdst> nested element is used to invoke the executable script `physicalfilerestore.sh` then restores the file object inside the save file into a specified library on the destination system.

IBM i Before you run this sample, you must complete some configuration as described in the `physicalfiletransfer.xml` file. You must also have a working Managed File Transfer environment on two IBM i systems. The setup must consist of a source agent running on the first IBM i system and a destination agent running on the second IBM i system. The two agents must be able to communicate with each other.

IBM i The physicalfiletransfer sample consists of the following three files:

- `physicalfiletransfer.xml`
- `physicalfilesave.sh` (<presrc> executable script)
- `physicalfilerestore.sh` (<postdst> executable script)

The sample files are located in the following directory: `/QIBM/ProdData/WMQFTE/V7/samples/fteant/ibmi/physicalfiletransfer`

IBM i To run this sample the user must complete the following steps:

1. Start a Qshell session. At an IBM i command window type: STRQSH
2. Change directory to the `bin` directory as follows:

```
cd  /QIBM/ProdData/WMQFTE/V7/bin
```

3. After completing the required configuration, run the sample by using the following command:

```
fteant -f /QIBM/ProdData/WMQFTE/V7/samples/fteant/ibmi/physicalfiletransfer/
physicalfiletransfer.xml
```

timeout

The timeout sample demonstrates how to use Ant tasks to attempt a file transfer and to cancel the transfer if it takes longer than a specified timeout value. The script initiates a file transfer by using the Managed File Transfer fte:filecopy task. The outcome of this transfer is deferred. The script uses the Managed File Transfer fte:awaitoutcome Ant task to wait a given number of seconds for the transfer to complete. If the transfer does not complete in the given time, the Managed File Transfer fte:cancel Ant task is used to cancel the file transfer.

vsamtransfer

z/OS

z/OS The vsamtransfer sample demonstrates how to use Ant tasks to transfer from a VSAM data set to another VSAM data set by using Managed File Transfer. Managed File Transfer currently does not support transferring VSAM data sets. The sample script unloads the VSAM data records to a sequential data set by using the `presrc` Program invocation nested elements to call the executable file `datasetcopy.sh`. The script uses the Managed File Transfer fte:filemove task to transfer the sequential data set from the source agent to the destination agent. The script then uses the `postdst` Program invocation nested elements to call the `loadvsam.jcl` script. This JCL script loads the transferred data set records into a destination VSAM data set. This sample uses JCL for the destination call to demonstrate this language option. The same result can also be achieved by using a second shell script instead.

z/OS This sample does not require the source and destination data sets to be VSAM. The sample works for any data sets if the source and destination data sets are of the same type.

z/OS For this sample to work correctly, you must put the `datasetcopy.sh` script on the command path of the source agent and the `loadvsam.jcl` script on the command path of the destination agent. For more information about setting the command path of an agent, see commandPath MFT property.

zip

The zip sample is made up of two scripts: `zip.xml` and `zipfiles.xml`. The sample demonstrates how to use the `presrc` nested element inside the Managed File Transfer fte:filemove task to run an Ant script before performing a file transfer move operation. The `zipfiles.xml` script called by the `presrc` nested element in the `zip.xml` script compresses the contents of a directory. The `zip.xml` script transfers the compressed file. This sample requires that the `zipfiles.xml` Ant script is present on the command path of the source agent. This is because the `zipfiles.xml` Ant script contains the target used to compress the contents of the directory at the source agent. For more information about setting the command path of an agent, see commandPath MFT property.

Related concepts

“Getting started using Ant scripts with MFT” on page 1178
Using Ant scripts with Managed File Transfer allows you to coordinate complex file transfer operations from an interpreted scripting language.

Related reference

fteAnt: run Ant tasks in MFT

Customizing MFT with user exits

You can customize the features of Managed File Transfer by using your own programs known as user exit routines.

Important: Any code within a user exit is not supported by IBM, and any issues with that code need to be initially investigated by either your enterprise, or the vendor who provided the exit.

Managed File Transfer provides points in the code where Managed File Transfer can pass control to a program that you have written (a user exit routine). These points are known as user exit points. Managed File Transfer can then resume control when your program has finished its work. You do not have to use any of the user exits, but they are useful if you want to extend and customize the function of your Managed File Transfer system to meet your specific requirements.

There are two points during file transfer processing where you can invoke a user exit at the source system and two points during file transfer processing where you can invoke a user exit at the destination system. The following table summarizes each of these user exit points and the Java interface that you must implement to use the exit points.

Table 187. Summary of source-side and destination-side exit points and Java interfaces

Exit point	**Java interface to implement**
Source-side exit points:	
Before the entire file transfer starts	SourceTransferStartExit.java interface
After the entire file transfer is complete	SourceTransferEndExit.java interface
Destination-side exit points:	
Before the entire file transfer starts	DestinationTransferStartExit.java interface
After the entire file transfer is complete	DestinationTransferEndExit.java interface

The user exits are invoked in the following order:

1. SourceTransferStartExit
2. DestinationTransferStartExit
3. DestinationTransferEndExit
4. SourceTransferEndExit

Changes made by the SourceTransferStartExit and DestinationTransferStartExit exits are propagated as input to subsequent exits. For example if the SourceTransferStartExit exit modifies the transfer metadata, the changes are reflected in the input transfer metadata to the other exits.

User exits and program calls are invoked in the following order:

```
- SourceTransferStartExit(onSourceTransferStart).
- PRE_SOURCE Command.
- DestinationTransferStartExits(onDestinationTransferStart).
- PRE_DESTINATION Command.
- The Transfer request is performed.
- DestinationTransferEndExits(onDestinationTransferENd).
- POST_DESTINATION Command.
- SourceTransferEndExits(onSourceTransferEnd.
- POST_SOURCE Command.
```

Notes:

1. The **DestinationTransferEndExits** is run only when the transfer completes, either successfully or partially successfully.
2. The **postDestinationCall** is run only when the transfer completes, either successfully or partially successfully.
3. The **SourceTransferEndExits** is run for successful, partially successful, or failed transfers.

4. The **postSourceCall** is called only if:
   - The transfer was not canceled.
   - There is a successful or partially successful outcome.
   - Any post-destination transfer programs ran successfully.

Building your user exit

The interfaces to build a user exit are contained in *MQ_INSTALL_DIRECTORY*/mqft/lib/com.ibm.wmqfte.exitroutines.api.jar. You must include this .jar file in the class path when you build your exit. To run the exit, extract the exit as a .jar file and place this .jar file in a directory as described in the following section.

User exit locations

You can store your user exit routines in two possible locations:

- The exits directory. There is an exits directory under each agent directory. For example: var\mqm\mqft\config\QM_JUPITER\agents\AGENT1\exits
- You can set the exitClassPath property to specify an alternative location. If there are exit classes in both the exits directory and the class path set by exitClassPath, the classes in the exits directory take priority, which means that if there are classes in both locations with the same name, the classes in the exits directory take priority.

Configuring an agent to use user exits

There are four agent properties that can be set to specify the user exits that an agent invokes. These agent properties are sourceTransferStartExitClasses, sourceTransferEndExitClasses, destinationTransferStartExitClasses, and destinationTransferEndExitClasses. For information about how to use these properties, see MFT Agent properties for user exits.

Running user exits on protocol bridge agents

When the source agent invokes the exit, it passes the exit a list of the source items for the transfer. For normal agents, this is a list of fully-qualified filenames. Because the files should be local (or accessible via a mount), then the exit is able to access it and encrypt it.

However, for a Protocol Bridge Agent, the entries in the list are of the following format:

```
"<file server identifier>:<fully-qualified file name of the file on the remote file server>"
```

For each entry in the list, the exit needs to connect to the file server first (using either the FTP. FTPS or SFTP protocols), download the file, encrypt it locally and then upload the encrypted file back to the file server.

Running user exits on Connect:Direct bridge agents

You cannot run user exits on Connect:Direct® bridge agents.

Related concepts

“MFT source and destination user exits” on page 1184

Metadata for MFT user exits

Java interfaces for MFT user exits

Related reference

“Enabling remote debugging for MFT user exits” on page 1188

While you are developing your user exits, you might want to use a debugger to help locate problems in your code.

“Sample MFT source transfer user exit” on page 1189

“Sample protocol bridge credential user exit” on page 1190

MFT resource monitor user exits
MFT Agent properties for user exits

MFT source and destination user exits

Directory separators

Directory separators in source file specifications are always represented using forward slash (/) characters, regardless of how you have specified directory separators in the **fteCreateTransfer** command or in the IBM MQ Explorer. You must take this into account when you write an exit. For example, if you want to check that the following source file exists: `c:\a\b.txt` and you have specified this source file using the **fteCreateTransfer** command or the IBM MQ Explorer, note the file name is actually stored as: `c:/a/b.txt` So if you search for the original string of `c:\a\b.txt`, you will not find a match.

Source side exit points

Before the entire file transfer starts

This exit is called by the source agent when a transfer request is next in the list of pending transfers and the transfer is about to start.

Example uses of this exit point are to send files in stages to a directory that the agent has read/write access to using an external command, or to rename the files on the destination system.

Pass the following arguments to this exit:

- Source agent name
- Destination agent name
- Environment metadata
- Transfer metadata
- File specifications (including file metadata)

The data returned from this exit is as follows:

- Updated transfer metadata. Entries can be added, modified, and deleted.
- Updated list of file specifications, which consists of source file name and destination file name pairs. Entries can be added, modified, and deleted
- Indicator that specifies whether to continue the transfer
- String to insert to the Transfer Log.

Implement the SourceTransferStartExit.java interface to call user exit code at this exit point.

After the entire file transfer is complete

This exit is called by the source agent after the entire file transfer has completed.

An example use of this exit point is to perform some completion tasks, such as sending an e-mail or an IBM MQ message to flag that the transfer has completed.

Pass the following arguments to this exit:

- Transfer exit result
- Source agent name
- Destination agent name
- Environment metadata
- Transfer metadata
- File results

The data returned from this exit is as follows:

- Updated string to insert to the Transfer Log.

Implement the SourceTransferEndExit.java interface to call user exit code at this exit point.

Destination side exit points

Before the entire file transfer starts

An example use of this exit point is to validate the permissions at the destination.

Pass the following arguments to this exit:

- Source agent name
- Destination agent name
- Environment metadata
- Transfer metadata
- File specifications

The data returned from this exit is as follows:

- Updated set of destination file names. Entries can be modified but not added or deleted.
- Indicator that specifies whether to continue the transfer
- String to insert into the Transfer Log.

Implement the DestinationTransferStartExit.java interface to call user exit code at this exit point.

After the entire file transfer is complete

An example use of this user exit is to start a batch process that uses the transferred files or to send an e-mail if the transfer has failed.

Pass the following arguments to this exit:

- Transfer exit result
- Source agent name
- Destination agent name
- Environment metadata
- Transfer metadata
- File results

The data returned from this exit is as follows:

- Updated string to insert to the Transfer Log.

Implement the DestinationTransferEndExit.java interface to call user exit code at this exit point.

Related concepts

Java interfaces for MFT user exits

Related reference

“Enabling remote debugging for MFT user exits” on page 1188

While you are developing your user exits, you might want to use a debugger to help locate problems in your code.

“Sample MFT source transfer user exit” on page 1189

MFT resource monitor user exits

Using MFT transfer I/O user exits

You can use Managed File Transfer transfer I/O user exits to configure custom code to perform the underlying file system I/O work for Managed File Transfer transfers.

Usually for MFT transfers, an agent selects from one of the built-in I/O providers to interact with the appropriate file systems for the transfer. Built-in I/O providers support the following types of file system:

- Regular UNIX-type and Windows-type file systems
- z/OS z/OS sequential and partitioned data sets (on z/OS only)
- IBM i IBM i native save files (on IBM i only)
- IBM MQ queues
- Remote FTP and SFTP protocol servers (for protocol bridge agents only)
- Remote Connect:Direct nodes (for Connect:Direct bridge agents only)

For file systems that are not supported, or where you require custom I/O behavior, you can write a transfer I/O user exit.

Transfer I/O user exits use the existing infrastructure for user exits. However, these transfer I/O user exits differ from other user exits because their function is accessed multiple times throughout the transfer for each file.

Use the agent property IOExitClasses (in the `agent.properties` file) to specify which I/O exit classes to load. Separate each exit class with a comma, for example:

```
IOExitClasses=testExits.TestExit1,testExits.testExit2
```

The Java interfaces for the transfer I/O user exits are as follows:

IOExit
The main entry point used to determine if the I/O exit is used. This instance is responsible for making IOExitPath instances.

You need specify only the IOExit I/O exit interface for the agent property IOExitClasses.

IOExitPath
Represents an abstract interface; for example, a data container or wildcard representing a set of data containers. You cannot create a class instance that implements this interface. The interface allows the path to be examined and derived paths to be listed. The IOExitResourcePath and IOExitWildcardPath interfaces extend IOExitPath.

IOExitChannel
Enables data to be read from or written to an IOExitPath resource.

IOExitRecordChannel
Extends the IOExitChannel interface for record-oriented IOExitPath resources, which enables data to be read from or written to an IOExitPath resource in multiples of records.

IOExitLock
Represents a lock on an IOExitPath resource for shared or exclusive access.

z/OS **IOExitRecordResourcePath**
Extends the IOExitResourcePath interface to represent a data container for a record-oriented file; for example, a z/OS data set. You can use the interface to locate data and to create IOExitRecordChannel instances for read or write operations.

IOExitResourcePath
Extends the IOExitPath interface to represent a data container; for example, a file or directory. You can use the interface to locate data. If the interface represents a directory, you can use the listPaths method to return a list of paths.

IOExitWildcardPath
Extends the IOExitPath interface to represent a path that denotes a wildcard. You can use this interface to match multiple IOExitResourcePaths.

IOExitProperties
Specifies properties that determine how Managed File Transfer handles IOExitPath for certain aspects of I/O. For example, whether to use intermediate files or whether to reread a resource from the beginning if a transfer is restarted.

Related concepts
"Customizing MFT with user exits" on page 1182
You can customize the features of Managed File Transfer by using your own programs known as user exit routines.

Related reference
IOExit.java interface
IOExitChannel.java interface
IOExitLock.java interface
IOExitPath.java interface
IOExitProperties.java interface
IOExitRecordChannel.java interface
z/OS IOExitRecordResourcePath.java interface
IOExitResourcePath.java interface
IOExitWildcardPath.java interface
The MFT`agent.properties` file

IBM i Sample MFT on IBM i user exits

Managed File Transfer provides sample user exits specific to IBM i with your installation. The samples are in the directories *MQMFT_install_dir*`/samples/ioexit-IBMi` and *MQMFT_install_dir*`/samples/userexit-IBMi`.

com.ibm.wmqfte.exit.io.ibmi.qdls.FTEQDLSExit
The com.ibm.wmqfte.exit.io.ibmi.qdls.FTEQDLSExit sample user exit transfers files in the QDLS file system on IBM i. After the exit is installed, any transfers to files that begin with `/QDLS` automatically use the exit.

To install this exit, complete the following steps:

1. Copy the `com.ibm.wmqfte.samples.ibmi.ioexits.jar` file from the *WMQFTE_install_dir*`/samples/ioexit-IBMi` directory to the agent's `exits` directory.
2. Add com.ibm.wmqfte.exit.io.ibmi.qdls.FTEQDLSExit to the IOExitClasses property.
3. Restart the agent.

com.ibm.wmqfte.exit.user.ibmi.FileMemberMonitorExit
The com.ibm.wmqfte.exit.user.ibmi.FileMemberMonitorExit sample user exit behaves like an MFT file monitor and automatically transfers physical file members from an IBM i library.

To run this exit, specify a value for the "library.qsys.monitor" metadata field (using the **-md** parameter, for example). This parameter takes an IFS-style path to a file member and can contain file and member wildcards. For example, /QSYS.LIB/FOO.LIB/BAR.FILE/*.MBR, /QSYS.LIB/FOO.LIB/*.FILE/BAR.MBR, /QSYS.LIB/FOO.LIB/*.FILE/*.MBR.

This sample exit also has an optional metadata field "naming.scheme.qsys.monitor", which you can use to determine the naming scheme that is used during the transfer. By default, this field is set to "unix,", which causes the destination file to be called F00.MBR. You can also specify the value "ibmi" to use the IBM i FTP FILE.MEMBER scheme, for example, /QSYS.LIB/FOO.LIB/BAR.FILE/BAZ.MBR is transferred as BAR.BAZ.

To install this exit, complete the following steps:

1. Copy the `com.ibm.wmqfte.samples.ibmi.userexits.jar` file from the *WMQFTE_install_dir*`/samples/userexit-IBMi` directory to the agent's `exits` directory.
2. Add com.ibm.wmqfte.exit.user.ibmi.FileMemberMonitorExit to the sourceTransferStartExitClasses property in the `agent.properties` file.
3. Restart the agent.

com.ibm.wmqfte.exit.user.ibmi.EmptyFileDeleteExit

The com.ibm.wmqfte.exit.user.ibmi.EmptyFileDeleteExit sample user exit deletes an empty file object when the source file member is deleted as part of the transfer. Because IBM i file objects can potentially hold many members, file objects are treated like directories by MFT. Therefore, you cannot perform a move operation on a file object using MFT; move operations are supported at the member level only. Consequently, when you perform a move operation on a member, the now empty file is left behind. Use this sample exit if you want to delete these empty files as part of the transfer request.

If you specify "true" for the "empty.file.delete" metadata and transfer an FTEFileMember, the sample exit deletes the parent file if the file is empty.

To install this exit, complete the following steps:

1. Copy the com.ibm.wmqfte.samples.ibmi.userexits.jar file from *WMQFTE_install_dir*`/samples/userexit-IBMi` to the agent's `exits` directory.
2. Add com.ibm.wmqfte.exit.user.ibmi.EmptyFileDeleteExit to the sourceTransferStartExitClasses property in the `agent.properties` file.
3. Restart the agent.

Related reference

“Using MFT transfer I/O user exits” on page 1186
You can use Managed File Transfer transfer I/O user exits to configure custom code to perform the underlying file system I/O work for Managed File Transfer transfers.

MFT Agent properties for user exits

Enabling remote debugging for MFT user exits

While you are developing your user exits, you might want to use a debugger to help locate problems in your code.

Because exits run inside the Java virtual machine that runs the agent, you cannot use the direct debugging support that is typically included in an integrated development environment. However, you can enable remote debugging of the JVM and then connect a suitable remote debugger.

To enable remote debugging, use the standard JVM parameters **-Xdebug** and **-Xrunjdwp**. These properties are passed to the JVM that runs the agent by the **BFG_JVM_PROPERTIES** environment variable. For example, on AIX and Linux the following commands start the agent and cause the JVM to listen for debugger connections on TCP port 8765.

```
export BFG_JVM_PROPERTIES="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,address=8765"
fteStartAgent -F TEST_AGENT
```

The agent does not start until the debugger connects. Use the **set** command on Windows instead of the **export** command.

You can also use other communication methods between the debugger and JVM. For example, the JVM can open the connection to the debugger instead of vice versa, or you can use shared memory instead of TCP. See the Java Platform Debugger Architecture documentation for further details.

You must use the **-F** (foreground) parameter when you start the agent in remote debug mode.

Using the Eclipse debugger

The following steps apply to the remote debugging capability in the Eclipse development environment. You can also use other remote debuggers that are JPDA-compatible.

1. Click **Run** > **Open Debug Dialog** (or **Run** > **Debug Configurations** or **Run** > **Debug Dialog** depending on your version of Eclipse).
2. Double-click **Remote Java Application** in the list of configuration types to create a debug configuration.
3. Complete the configuration fields and save the debug configuration. If you have already started the agent JVM in debug mode, you can connect to the JVM now.

Sample MFT source transfer user exit

```
/*
 * A Sample Source Transfer End Exit that prints information about a transfer to standard
output.
 * If the agent is run in the background the output will be sent to the agent's event log file.
If
 * the agent is started in the foreground by specifying the -F parameter on the fteStartAgent
 * command the output will be sent to the console.
 *
 * To run the exit execute the following steps:
 *
 * Compile and build the exit into a jar file. You need the following in the class path:
 * {MQ_INSTALLATION_PATH}\mqft\lib\com.ibm.wmqfte.exitroutines.api.jar
 *
 * Put the jar in your agent's exits directory:
 * {MQ_DATA_PATH}\config\coordQmgrName\agents\agentName\exits\
 *
 * Update the agent's properties file:
 * {MQ_DATA_PATH}\config\coordQmgrName\agents\agentName\agent.properties
 * to include the following property:
 * sourceTransferEndExitClasses=[packageName.]SampleEndExit
 *
 * Restart agent to pick up the exit
 *
 * Send the agent a transfer request:
 * For example: fteCreateTransfer -sa myAgent -da YourAgent -df output.txt input.txt
 */

import java.util.List;
import java.util.Map;
import java.util.Iterator;

import com.ibm.wmqfte.exitroutine.api.SourceTransferEndExit;
import com.ibm.wmqfte.exitroutine.api.TransferExitResult;
import com.ibm.wmqfte.exitroutine.api.FileTransferResult;

public class SampleEndExit implements SourceTransferEndExit {

    public String onSourceTransferEnd(TransferExitResult transferExitResult,
            String sourceAgentName,
            String destinationAgentName,
            Map<String, String>environmentMetaData,
            Map<String, String>transferMetaData,
            List<FileTransferResult>fileResults) {

        System.out.println("Environment Meta Data: " + environmentMetaData);
        System.out.println("Transfer Meta Data:    " + transferMetaData);

        System.out.println("Source agent:          " +
                sourceAgentName);
        System.out.println("Destination agent:     " +
                destinationAgentName);
```

```
        if (fileResults.isEmpty()) {
            System.out.println("No files in the list");
            return "No files";
        }
        else {

            System.out.println( "File list: ");

            final Iterator<FileTransferResult> iterator = fileResults.iterator();

            while (iterator.hasNext()){
                final FileTransferResult thisFileSpec = iterator.next();
                System.out.println("Source file spec: "  +
                        thisFileSpec.getSourceFileSpecification() +
                        ", Destination file spec: " +
                        thisFileSpec.getDestinationFileSpecification());
            }
        }
        return "Done";
    }
}
```

Sample protocol bridge credential user exit

For information about how to use this sample user exit, see Mapping credentials for a file server by using exit classes.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.StringTokenizer;

import com.ibm.wmqfte.exitroutine.api.CredentialExitResult;
import com.ibm.wmqfte.exitroutine.api.CredentialExitResultCode;
import com.ibm.wmqfte.exitroutine.api.CredentialPassword;
import com.ibm.wmqfte.exitroutine.api.CredentialUserId;
import com.ibm.wmqfte.exitroutine.api.Credentials;
import com.ibm.wmqfte.exitroutine.api.ProtocolBridgeCredentialExit;

/**
 * A sample protocol bridge credential exit
 *
 * This exit reads a properties file that maps mq user ids to server user ids
 * and server passwords. The format of each entry in the properties file is:
 *
 * mqUserId=serverUserId,serverPassword
 *
 * The location of the properties file is taken from the protocol bridge agent
 * property protocolBridgeCredentialConfiguration.
 *
 * To install the sample exit compile the class and export to a jar file.
 * Place the jar file in the exits subdirectory of the agent data directory
 * of the protocol bridge agent on which the exit is to be installed.
 * In the agent.properties file of the protocol bridge agent set the
 * protocolBridgeCredentialExitClasses to SampleCredentialExit
 * Create a properties file that contains the mqUserId to serverUserId and
 * serverPassword mappings applicable to the agent. In the agent.properties
 * file of the protocol bridge agent set the protocolBridgeCredentialConfiguration
 * property to the absolute path name of this properties file.
 * To activate the changes stop and restart the protocol bridge agent.
 *
 * For further information on protocol bridge credential exits refer to
 * the WebSphere MQ Managed File Transfer documentation online at:
 * https://www.ibm.com/docs/SSEP7X_7.0.4/welcome/WelcomePagev7r0.html
 */
public class SampleCredentialExit implements ProtocolBridgeCredentialExit {

    // The map that holds mq user ID to serverUserId and serverPassword mappings
    final private Map<String,Credentials> credentialsMap = new HashMap<String, Credentials>();
```

```
    /* (non-Javadoc)
      * @see com.ibm.wmqfte.exitroutine.api.ProtocolBridgeCredentialExit#initialize(java.util.Map)
      */
     public synchronized boolean initialize(Map<String, String> bridgeProperties) {

        // Flag to indicate whether the exit has been successfully initialized or not
        boolean initialisationResult = true;

        // Get the path of the mq user ID mapping properties file
         final String propertiesFilePath = bridgeProperties.get("protocolBridgeCredentialConfiguration");

         if (propertiesFilePath == null || propertiesFilePath.length() == 0) {
             // The properties file path has not been specified. Output an error and return false
             System.err.println("Error initializing SampleCredentialExit.");
              System.err.println("The location of the mqUserID mapping properties file has not been
specified in the
       protocolBridgeCredentialConfiguration property");
             initialisationResult = false;
         }

        if (initialisationResult) {

            // The Properties object that holds mq user ID to serverUserId and serverPassword
             // mappings from the properties file
             final Properties mappingProperties = new Properties();

             // Open and load the properties from the properties file
             final File propertiesFile = new File (propertiesFilePath);
             FileInputStream inputStream = null;
             try {
                 // Create a file input stream to the file
                 inputStream = new FileInputStream(propertiesFile);

                 // Load the properties from the file
                 mappingProperties.load(inputStream);
             }
             catch (FileNotFoundException ex) {
                 System.err.println("Error initializing SampleCredentialExit.");
                  System.err.println("Unable to find the mqUserId mapping properties file: " +
propertiesFilePath);
                 initialisationResult = false;
             }
             catch (IOException ex) {
                 System.err.println("Error initializing SampleCredentialExit.");
                  System.err.println("Error loading the properties from the mqUserId mapping properties
file: " + propertiesFilePath);
                 initialisationResult = false;
             }
             finally {
                 // Close the inputStream
                 if (inputStream != null) {
                     try {
                         inputStream.close();
                     }
                     catch (IOException ex) {
                         System.err.println("Error initializing SampleCredentialExit.");
                          System.err.println("Error closing the mqUserId mapping properties file: " +
propertiesFilePath);
                         initialisationResult = false;
                     }
                 }
             }

             if (initialisationResult) {
                 // Populate the map of mqUserId to server credentials from the properties
                 final Enumeration<?> propertyNames = mappingProperties.propertyNames();
                  while ( propertyNames.hasMoreElements()) {
                     final Object name = propertyNames.nextElement();
                     if (name instanceof String ) {
                         final String mqUserId = ((String)name).trim();
                          // Get the value and split into serverUserId and serverPassword
                        final String value = mappingProperties.getProperty(mqUserId);
                        final StringTokenizer valueTokenizer = new StringTokenizer(value, ",");
                         String serverUserId = "";
                         String serverPassword = "";
                         if (valueTokenizer.hasMoreTokens()) {
                             serverUserId = valueTokenizer.nextToken().trim();
                         }
                         if (valueTokenizer.hasMoreTokens()) {
                             serverPassword = valueTokenizer.nextToken().trim();
                         }
```

```
                        // Create a Credential object from the serverUserId and serverPassword
final Credentials credentials = new Credentials(new CredentialUserId(serverUserId), new
CredentialPassword(serverPassword));
                        // Insert the credentials into the map
                        credentialsMap.put(mqUserId, credentials);
                    }
                }

            }
        }

        return initialisationResult;
    }
     /* (non-Javadoc)
     * @see com.ibm.wmqfte.exitroutine.api.ProtocolBridgeCredentialExit#mapMQUserId(java.lang.String)
     */
    public synchronized CredentialExitResult mapMQUserId(String mqUserId) {
        CredentialExitResult result = null;
        // Attempt to get the server credentials for the given mq user id
        final Credentials credentials = credentialsMap.get(mqUserId.trim());
        if ( credentials == null) {
            // No entry has been found so return no mapping found with no credentials
            result = new CredentialExitResult(CredentialExitResultCode.NO_MAPPING_FOUND, null);
        }
        else {
            // Some credentials have been found so return success to the user along with the credentials
            result = new CredentialExitResult(CredentialExitResultCode.USER_SUCCESSFULLY_MAPPED,
credentials);
        }
        return result;
    }
     /* (non-Javadoc)
     * @see com.ibm.wmqfte.exitroutine.api.ProtocolBridgeCredentialExit#shutdown(java.util.Map)
     */
    public void shutdown(Map<String, String> bridgeProperties) {
        // Nothing to do in this method because there are no resources that need to be released
    }
}
```

Sample protocol bridge properties user exit

For information about how to use this sample user exit, see ProtocolBridgePropertiesExit2: Looking up protocol file server properties

SamplePropertiesExit2.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Properties;

import com.ibm.wmqfte.exitroutine.api.ProtocolBridgePropertiesExit2;
import com.ibm.wmqfte.exitroutine.api.ProtocolServerPropertyConstants;

/**
 * A sample protocol bridge properties exit. This exit reads a properties file
 * that contains properties for protocol servers.
 * <p>
 * The format of each entry in the properties file is:
 * {@literal serverName=type://host:port}
 * Ensure there is a default entry such as
 * {@literal default=type://host:port}
 * otherwise the agent will fail to start with a BFGBR0168 as it must have a
 * default server.
 * <p>
 * The location of the properties file is taken from the protocol bridge agent
 * property {@code protocolBridgePropertiesConfiguration}.
 * <p>
 * The methods {@code getCredentialLocation} returns the location of the associated
 * ProtocolBridgeCredentials.xml, this sample it is defined to be stored in a directory
 * defined by the environment variable CREDENTIALSHOME
 * <p>
```

```
 * To install the sample exit:
 * <ol>
 * <li>Compile the class and export to a jar file.
 * <li>Place the jar file in the {@code exits} subdirectory of the agent data directory
 * of the protocol bridge agent on which the exit is to be installed.
 * <li>In the {@code agent.properties} file of the protocol bridge agent
 * set the {@code protocolBridgePropertiesExitClasses} to
 * {@code SamplePropertiesExit2}.
 * <li>Create a properties file that contains the appropriate properties to specify the
 * required servers.
 * <li>In the {@code agent.properties} file of the protocol bridge agent
 * set the <code>protocolBridgePropertiesConfiguration</code> property to the
 * absolute path name of this properties file.
 * <li>To activate the changes stop and restart the protocol bridge agent.
 * </ol>
 * <p>
 * For further information on protocol bridge properties exits refer to the
 * WebSphere MQ Managed File Transfer documentation online at:
 * <p>
 * {@link https://www.ibm.com/docs/SSEP7X_7.0.4/welcome/WelcomePagev7r0.html}
 */
public class SamplePropertiesExit2 implements ProtocolBridgePropertiesExit2 {

    /**
     * Helper class to encapsulate protocol server information.
     */
    private static class ServerInformation {
        private final String type;
        private final String host;
        private final int port;

        public ServerInformation(String url) {
            int index = url.indexOf("://");
            if (index == -1) throw new IllegalArgumentException("Invalid server URL: "+url);
            type = url.substring(0, index);

            int portIndex = url.indexOf(":", index+3);
            if (portIndex == -1) {
                host = url.substring(index+3);
                port = -1;
            } else {
                host = url.substring(index+3,portIndex);
                port = Integer.parseInt(url.substring(portIndex+1));
            }
        }

        public String getType() {
            return type;
        }

        public String getHost() {
            return host;
        }

        public int getPort() {
            return port;
        }
    }

    /** A {@code Map} that holds information for each configured protocol server */
    final private Map<String, ServerInformation> servers = new HashMap<String, ServerInformation>();


    /* (non-Javadoc)
     * @see
com.ibm.wmqfte.exitroutine.api.ProtocolBridgePropertiesExit#getProtocolServerProperties(java.lang.String)
     */
    public Properties getProtocolServerProperties(String protocolServerName) {
        // Attempt to get the protocol server information for the given protocol server name
        // If no name has been supplied then this implies the default.
        final ServerInformation info;
        if (protocolServerName == null || protocolServerName.length() == 0) {
            protocolServerName = "default";
        }
        info = servers.get(protocolServerName);

        // Build the return set of properties from the collected protocol server information, when
available.
        // The properties set here is the minimal set of properties to be a valid set.
        final Properties result;
        if (info != null) {
            result = new Properties();
```

```
            result.setProperty(ProtocolServerPropertyConstants.SERVER_NAME, protocolServerName);
            result.setProperty(ProtocolServerPropertyConstants.SERVER_TYPE, info.getType());
            result.setProperty(ProtocolServerPropertyConstants.SERVER_HOST_NAME, info.getHost());
            if (info.getPort() != -1)
result.setProperty(ProtocolServerPropertyConstants.SERVER_PORT_VALUE, ""+info.getPort());
            result.setProperty(ProtocolServerPropertyConstants.SERVER_PLATFORM, "UNIX");
            if (info.getType().toUpperCase().startsWith("FTP")) {        // FTP & FTPS
                result.setProperty(ProtocolServerPropertyConstants.SERVER_TIMEZONE, "Europe/London");
                result.setProperty(ProtocolServerPropertyConstants.SERVER_LOCALE, "en-GB");
            }
            result.setProperty(ProtocolServerPropertyConstants.SERVER_FILE_ENCODING, "UTF-8");
        } else {
            System.err.println("Error no default protocol file server entry has been supplied");
            result = null;
        }

        return result;
    }

    /* (non-Javadoc)
     * @see com.ibm.wmqfte.exitroutine.api.ProtocolBridgePropertiesExit#initialize(java.util.Map)
     */
    public boolean initialize(Map<String, String> bridgeProperties) {
        // Flag to indicate whether the exit has been successfully initialized or not
        boolean initialisationResult = true;

        // Get the path of the properties file
        final String propertiesFilePath = bridgeProperties.get("protocolBridgePropertiesConfiguration");
        if (propertiesFilePath == null || propertiesFilePath.length() == 0) {
            // The protocol server properties file path has not been specified. Output an error and
return false
            System.err.println("Error initializing SamplePropertiesExit.");
            System.err.println("The location of the protocol server properties file has not been
specified in the
      protocolBridgePropertiesConfiguration property");
            initialisationResult = false;
        }

        if (initialisationResult) {
            // The Properties object that holds protocol server information
            final Properties mappingProperties = new Properties();

            // Open and load the properties from the properties file
            final File propertiesFile = new File (propertiesFilePath);
            FileInputStream inputStream = null;
            try {
                // Create a file input stream to the file
                inputStream = new FileInputStream(propertiesFile);

                // Load the properties from the file
                mappingProperties.load(inputStream);
            } catch (final FileNotFoundException ex) {
                System.err.println("Error initializing SamplePropertiesExit.");
                System.err.println("Unable to find the protocol server properties file: " +
propertiesFilePath);
                initialisationResult = false;
            } catch (final IOException ex) {
                System.err.println("Error initializing SamplePropertiesExit.");
                System.err.println("Error loading the properties from the protocol server properties
file: " + propertiesFilePath);
                initialisationResult = false;
            } finally {
                // Close the inputStream
                if (inputStream != null) {
                    try {
                        inputStream.close();
                    } catch (final IOException ex) {
                        System.err.println("Error initializing SamplePropertiesExit.");
                        System.err.println("Error closing the protocol server properties file: " +
propertiesFilePath);
                        initialisationResult = false;
                    }
                }
            }

            if (initialisationResult) {
                // Populate the map of protocol servers from the properties
                for (Entry<Object, Object> entry : mappingProperties.entrySet()) {
                    final String serverName = (String)entry.getKey();
                    final ServerInformation info = new ServerInformation((String)entry.getValue());
                    servers.put(serverName, info);
                }
```

```
            }
        }

        return initialisationResult;
    }

    /* (non-Javadoc)
     * @see com.ibm.wmqfte.exitroutine.api.ProtocolBridgePropertiesExit#shutdown(java.util.Map)
     */
    public void shutdown(Map<String, String> bridgeProperties) {
        // Nothing to do in this method because there are no resources that need to be released
    }

    /* (non-Javadoc)
     * @see com.ibm.wmqfte.exitroutine.api.ProtocolBridgePropertiesExit2#getCredentialLocation()
     */
    public String getCredentialLocation() {
        String envLocationPath;
        if (System.getProperty("os.name").toLowerCase().contains("win")) {
            // Windows style
            envLocationPath = "%CREDENTIALSHOME%\\ProtocolBridgeCredentials.xml";
        }
        else {
            // Unix style
            envLocationPath = "$CREDENTIALSHOME/ProtocolBridgeCredentials.xml";
        }
        return envLocationPath;
    }

}
```

Controlling MFT by putting messages on the agent command queue

You can write an application that controls Managed File Transfer by putting messages on agent command queues.

You can put a message on the command queue of an agent to request that the agent performs one of the following actions:

- Create a file transfer
- Create a scheduled file transfer
- Cancel a file transfer
- Cancel a scheduled file transfer
- Call a command
- Create a monitor
- Delete a monitor
- Return a ping to indicate that the agent is active

To request that the agent performs one of these actions, the message must be in an XML format that complies with one of the following schema:

FileTransfer.xsd
: Messages in this format can be used to create a file transfer or scheduled file transfer, to call a command, or to cancel a file transfer or scheduled file transfer. For more information, see File transfer request message format.

Monitor.xsd
: Messages in this format can be used to create or delete a resource monitor. For more information, see MFT monitor request message formats.

PingAgent.xsd
: Messages in this format can be used to ping an agent to check that it is active. For more information, see Ping MFT agent request message format.

The agent returns a reply to the request messages. The reply message is put to a reply queue that is defined in the request message. The reply message is in an XML format defined by the following schema:

Reply.xsd
For more information, see MFT agent reply message format.

Developing applications for MQ Telemetry

Telemetry applications integrate sense and control devices with other sources of information available on the internet and in enterprises.

Develop applications for MQ Telemetry using design patterns, worked examples, sample programs, programming concepts, and reference information.

Related concepts

MQ Telemetry

Telemetry use cases

Related tasks

Installing MQ Telemetry

Administering MQ Telemetry

Troubleshooting MQ Telemetry problems

Related reference

MQ Telemetry Reference

IBM MQ Telemetry Transport sample programs

Sample scripts are provided that work with a sample IBM MQ Telemetry Transport v3 client application (`mqttv3app.jar`). For IBM MQ 8.0.0 and later, the sample client application is no longer included in MQ Telemetry. It was part of the (no longer available) IBM Messaging Telemetry Clients SupportPac. Similar sample applications continue to be freely available from Eclipse Paho and MQTT.org.

For the latest information and downloads, see the following resources:

- The Eclipse Paho project, and MQTT.org, have free downloads of the latest telemetry clients and samples for a range of programming languages. Use these sites to help you develop sample programs for publishing and subscribing IBM MQ Telemetry Transport, and for adding security features.
- The IBM Messaging Telemetry Clients SupportPac is no longer available for download. If you have a previously downloaded copy, it has the following contents:
  - The MA9B version of the IBM Messaging Telemetry Clients SupportPac included a compiled sample application (`mqttv3app.jar`) and associated client library (`mqttv3.jar`). They were provided in the following directories:
    - `ma9b/SDK/clients/java/org.eclipse.paho.sample.mqttv3app.jar`
    - `ma9b/SDK/clients/java/org.eclipse.paho.client.mqttv3.jar`
  - In the MA9C version of this SupportPac, the `/SDK/` directory and contents was removed:
    - Only the source for the sample application (`mqttv3app.jar`) was provided. It was in this directory:

      ```
      ma9c/clients/java/samples/org/eclipse/paho/sample/mqttv3app/*.java
      ```

    - The compiled client library was still provided. It was in this directory:

      ```
      ma9c/clients/java/org.eclipse.paho.client.mqttv3-1.0.2.jar
      ```

If you still have a copy of the (no longer available) IBM Messaging Telemetry Clients SupportPac, information about installing and running the sample application is provided in Verifying the installation of MQ Telemetry using the command line.

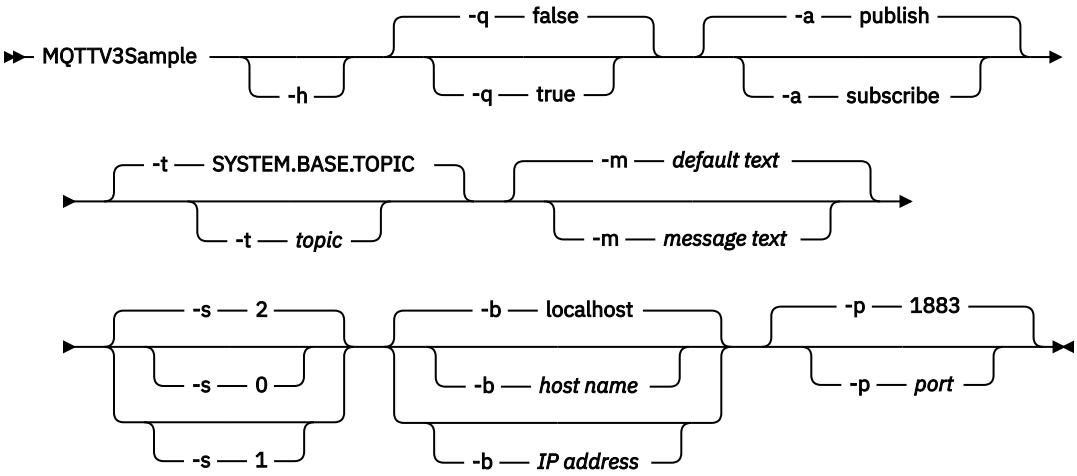
MQTTV3Sample program

Reference information about sample syntax and parameters for the MQTTV3Sample program.

Purpose

The MQTTV3Sample program can be used to publish a message and subscribe to a topic. For information about how to get this sample program, see “IBM MQ Telemetry Transport sample programs” on page 1196.

MQTTV3Sample syntax



Parameters

-h
Print this help text and quit

-q
Set quiet mode, instead of using the default mode of false.

-a
Set publish or subscribe, instead of assuming the default action of publishing.

-t
Publish or subscribe to topic, instead of publishing or subscribing to the default topic

-m
Publish message text instead of sending the default publication text, "Hello from an MQTT v3 application".

-s
Set QoS instead of using the default QoS, 2.

-b
Connect to this host name or IP address instead of connecting to the default host name, localhost.

-p
Use this port instead of using the default, 1883.

Run the MQTTV3Sample program

To subscribe to a topic on Windows, use the command:

```
run MQTTV3Sample -a subscribe
```

To publish a message on Windows, use the command:

```
run MQTTV3Sample
```

MQTT client programming concepts

The concepts described in this section help you to understand the client libraries for the MQTT protocol. The concepts complement the API documentation accompanying the client libraries.

For the latest information and downloads, see the following resources:

- The Eclipse Paho project, and MQTT.org, have free downloads of the latest telemetry clients and samples for a range of programming languages. Use these sites to help you develop sample programs for publishing and subscribing IBM MQ Telemetry Transport, and for adding security features.
- The IBM Messaging Telemetry Clients SupportPac is no longer available for download. If you have a previously downloaded copy, it has the following contents:
  - The MA9B version of the IBM Messaging Telemetry Clients SupportPac included a compiled sample application (`mqttv3app.jar`) and associated client library (`mqttv3.jar`). They were provided in the following directories:
    - `ma9b/SDK/clients/java/org.eclipse.paho.sample.mqttv3app.jar`
    - `ma9b/SDK/clients/java/org.eclipse.paho.client.mqttv3.jar`
  - In the MA9C version of this SupportPac, the `/SDK/` directory and contents was removed:
    - Only the source for the sample application (`mqttv3app.jar`) was provided. It was in this directory:

      ```
      ma9c/clients/java/samples/org/eclipse/paho/sample/mqttv3app/*.java
      ```

    - The compiled client library was still provided. It was in this directory:

      ```
      ma9c/clients/java/org.eclipse.paho.client.mqttv3-1.0.2.jar
      ```

To develop and run an MQTT client you need to copy or install these resources on the client device. You do not need to install a separate client runtime.

The licensing conditions for clients are associated with the server that you are connecting the clients to.

The MQTT client libraries are reference implementations of the MQTT protocol. You can implement your own clients in different languages suitable for different device platforms. See IBM MQ Telemetry Transport format and protocol.

The API documentation makes no assumptions about which MQTT server the client is connected to. The behavior of the client might differ slightly when connected to different servers. The descriptions that follow describe the behavior of the client when connected to the IBM MQ telemetry service.

Callbacks and synchronization in MQTT client applications

The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Callbacks

Note: See the Eclipse Paho website for the latest changes to `MqttCallback`. For example `MqttCallback` is defined as an Interface in the Paho version of the client, and asynchronous methods are provided by the Paho `MqttAsyncClient` class.

The `MqttCallback` interface has three callback methods:

connectionLost(java.lang.Throwable cause)

connectionLost is called when a communications error leads to the connection dropping. It is also called if the server drops the connection as a result of an error on the server after the connection has been established. Server errors are logged to the queue manager error log. The server drops the connection to the client, and the client calls MqttCallback.connectionLost.

The only remote errors thrown as exceptions on the same thread as the client application are exceptions from MqttClient.connect. Errors detected by the server after the connection is established are reported back to the MqttCallback.connectionLost callback method as throwables.

Typical server errors that result in connectionLost are authorization errors. For example, the telemetry server tries to publish on a topic on behalf of a client that is not authorized to publish on the topic. Anything that results in a MQCC_FAIL condition code being returned to the telemetry server can result in the connection being dropped.

deliveryComplete(IMqttDeliveryToken token)

deliveryComplete is called by the MQTT client to pass a delivery token back to the client application; see "Delivery tokens" on page 1205. Using the delivery token, the callback can access the published message with the method token.getMessage.

When the application callback returns control to the MQTT client after being called by the deliveryComplete method, delivery is completed. Until delivery is completed, messages with QoS 1 or 2 are retained by the persistence class.

The call to deliveryComplete is a point of synchronization between the application and the persistence class. The deliveryComplete method is never called twice for the same message.

When the application callback returns from deliveryComplete to the MQTT client, the client calls MqttClientPersistence.remove for messages with QoS 1 or 2. MqttClientPersistence.remove deletes the locally stored copy of the published message.

From a transaction processing perspective, the call to deliveryComplete is a single phase transaction that commits the delivery. If processing fails during the callback, on restart of the client MqttClientPersistence.remove is called again to delete the local copy of the published message. The callback is not called again. If you are using the callback to store a log of delivered messages, you cannot synchronize the log with the MQTT client. If you want to store a log reliably, then update the log in the MqttClientPersistence class.

The delivery token and message are referenced by the main application thread and the MQTT client. The MQTT client dereferences the MqttMessage object when delivery is completed, and the delivery token object when the client disconnects. The MqttMessage object can be garbage collected after delivery is completed if the client application dereferences it. The delivery token can be garbage collected after the session is disconnected.

You can get IMqttDeliveryToken and MqttMessage attributes after a message has been published. If you attempt to set any MqttMessage attributes after the message has been published the result is undefined.

The MQTT client continues to process delivery acknowledgments if the client reconnects to the previous session with the same ClientIdentifier ; see "Clean sessions" on page 1202. The MQTT client application must set MqttClient.CleanSession to false for the previous session, and set it to false in the new session. The MQTT client creates new delivery tokens and message objects in the new session for pending deliveries. It recovers the objects using the MqttClientPersistence class. If the application client still has references to the old delivery tokens and messages, dereference them. The application callback is called in the new session for any deliveries initiated in the previous session and completed in this session.

The application callback is called after the application client connects, when a pending delivery is completed. Before the application client connects, it can retrieve pending deliveries using the MqttClient.getPendingDeliveryTokens method.

Notice that the client application originally created the message object that is published, and its payload byte array. The MQTT client references these objects. The message object returned by the delivery token in the method token.getMessage is not necessarily the same message object created by the client. If a new MQTT client instance re-creates the delivery token, the MqttClientPersistence class re-creates the MqttMessage object. For consistency

`token.getMessage` returns `null` if `token.isCompleted` is `true`, regardless of whether the message object was created by the application client or the `MqttClientPersistence` class.

`messageArrived(String topic, MqttMessage message)`

`messageArrived` is called when a publication arrives for the client that matches a subscription topic. `topic` is the publication topic, not the subscription filter. The two can be different if the filter contains wildcards.

If the topic matches multiple subscriptions created by the client, the client receives multiple copies of the publication. If a client publishes to a topic that it also subscribes to, it receives a copy of its own publication.

If a message is sent with a QoS of 1 or 2, the message is stored by the `MqttClientPersistence` class before the MQTT client calls `messageArrived`. `messageArrived` behaves like `deliveryComplete`: it is only called once for a publication, and the local copy of the publication is removed by `MqttClientPersistence.remove` when `messageArrived` returns to the MQTT client. The MQTT client drops its references to the topic and message when `messageArrived` returns to the MQTT client. The topic and message objects are garbage collected, if the application client has not held onto a reference to the objects.

Callbacks, threading, and client application synchronization

The MQTT client calls a callback method on a separate thread to the main application thread. The client application does not create a thread for the callback, it is created by the MQTT client.

The MQTT client synchronizes callback methods. Only one instance of the callback method runs at a time. The synchronization makes it easy to update an object that tallies which publications have been delivered. One instance of the `MqttCallback.deliveryComplete` runs at a time, and so it is safe to update the tally without further synchronization. It is also the case that only one publication arrives at a time. Your code in the `messageArrived` method can update an object without synchronizing it. If you are referring to the tally, or the object that is being updated, in another thread, synchronize the tally or object.

The delivery token provides a synchronization mechanism between the main application thread and delivery of a publication. The method `token.waitForCompletion` waits until delivery of a specific publication is completed, or until an optional timeout expires. You might use `token.waitForCompletion` in the following way to process one publication at a time.

To synchronize with the `MqttCallback.deliveryComplete` method. Only when `MqttCallback.deliveryComplete` returns to the MQTT Client does `token.waitForCompletion` resume. Using this mechanism you can synchronize running code in `MqttCallback.deliveryComplete` before code runs in the main application thread.

What if you wanted to publish without waiting for each publication to be delivered, but want confirmation when all the publications have been delivered? If you publish on a single thread, the last publication to be sent is also the last to be delivered.

Synchronization of requests sent to the server

Table 188 on page 1201 describes the methods in the MQTT Java client that send a request to the server. Unless the application client sets an indefinite timeout, the client never waits indefinitely for the server. If the client hangs, it is either an application programming problem, or a defect in the MQTT client.

Table 188. Synchronization behavior of methods that result in requests to the server		
Method	**Synchronization**	**Timeout interval**
`MqttClient.Connect`	Waits for a connection to be established with the server.	Defaults to 30 seconds, or as set by a parameter, then throws an exception.
`MqttClient.Disconnect`	Waits for the MQTT client to finish any work it must do, and for the TCP/IP session to disconnect.	
`MqttClient.Subscribe`	Waits for completion of the Subscribe or UnSubscribe method.	
`MqttClient.UnSubscribe`		
`MqttClient.Publish`	Returns immediately to the application thread after passing the request to the MQTT client.	None.
`IMqttDeliveryToken.waitFor Completion`	Waits for the delivery token to be returned.	Indefinite, or as set as a parameter.

Related concepts

Clean sessions

The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier

The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication

If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients

Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications

Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client

An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients

A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Clean sessions

The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

When you connect an MQTT client application using the `MqttClient.connect` method, the client identifies the connection using the client identifier and the address of the server. The server checks whether session information has been saved from a previous connection to the server. If a previous session still exists, and `cleanSession=true`, then the previous session information at the client and server is cleared. If `cleanSession=false` the previous session is resumed. If no previous session exists, a new session is started.

Note: The IBM MQ Administrator can forcibly close an open session and delete all the session information. If the client reopens the session with `cleanSession=false`, a new session is started.

Publications

If you use the default `MqttConnectOptions`, or set `MqttConnectOptions.cleanSession` to `true` before connecting the client, all pending publication deliveries for the client are removed when the client connects.

The clean session setting has no effect on publications sent with `QoS=0`. For `QoS=1` and `QoS=2`, using `cleanSession=true` might result in losing a publication.

Subscriptions

If you use the default `MqttConnectOptions`, or set `MqttConnectOptions.cleanSession` to `true` before connecting the client, any old subscriptions for the client are removed when the client connects. Any new subscriptions the client makes during the session are removed when it disconnects.

If you set `MqttConnectOptions.cleanSession` to `false` before connecting, any subscriptions the client creates are added to all the subscriptions that existed for the client before it connected. All the subscriptions remain active when the client disconnects.

Another way of understanding the way the `cleanSession` attribute affects subscriptions is to think of it as a modal attribute. In its default mode, `cleanSession=true`, the client creates subscriptions and receives publications only within the scope of the session. In the alternative mode, `cleanSession=false`, subscriptions are durable. The client can connect and disconnect and its subscriptions remain active. When the client reconnects, it receives any undelivered publications. While it is connected, it can modify the set of subscriptions that are active on its behalf.

You must set the `cleanSession` mode before connecting; the mode lasts for the whole session. To change its setting, you must disconnect and reconnect the client. If you change modes from using `cleanSession=false` to `cleanSession=true`, all previous subscriptions for the client, and any publications that have not been received, are discarded.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Client identifier

The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

The client identifier is used in the administration of an MQTT system. With potentially hundreds of thousands of clients to administer, you need to be able to identify a particular client rapidly. For example, suppose a device has malfunctioned and you are notified, perhaps by a customer ringing a help desk. The customer needs to be able to identify the device, and you need to be able to correlate that identification with the server that is typically connected to the client.

When you browse through MQTT client connections, each connection is labeled with the client identifier. To help decide how best to map this identifier to the device and server, ask yourself the following questions:

- Would it be convenient to maintain and use a database that maps each device to a client identifier and to a server?
- Could the name of the device identify the server to which it is attached?
- Do you need a look-up table that maps a client identifier to a physical device?
- Does the client identifier identify a particular device, a user, or an application running at the client?
- If a customer replaces a faulty device with a new one, does the new device have the same identifier as the old device, or do you allocate a new identifier? (If you change a physical device and keep the same identifier, outstanding publications and active subscriptions are automatically transferred to the new device.)

You also need a system to ensure that client identifiers are unique, and you must have a reliable process for setting the identifier on the client. If the client device is a "black-box", with no user interface, you could manufacture the device with a client identifier, or you could have a software installation and configuration process that configures the device before it is activated.

To keep the identifier short and unique, you could create a client identifier from the 48 bit device MAC address. If transmission size is not a critical issue, you could then use the remaining 17 bytes to make the address easier to administer.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Delivery tokens

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients

A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Delivery tokens

When a client publishes on a topic a new delivery token is created. Use the delivery token to monitor the delivery of a publication, or to block the client application until delivery is complete.

The token is an `MqttDeliveryToken` object. It is created by calling the MqttTopic.publish() method and is retained by the MQTT client until the client session is disconnected and the delivery is completed.

The normal use of the token is to check whether delivery is complete. Block the client application until delivery is complete by using the returned token to call `token.waitForCompletion`. Alternatively, provide a `MqttCallBack` handler. When the MQTT client has received all the acknowledgments it expects as part of delivering the publication, it calls `MqttCallBack.deliveryComplete` passing the delivery token as a parameter.

Until delivery is complete, you can inspect the publication using the returned delivery token by calling `token.getMessage`.

Completed deliveries

The completion of deliveries is asynchronous and depends on the quality of service associated with the publication.

At most once

`QoS=0`
Delivery is complete immediately on return from `MqttTopic.publish`. `MqttCallback.deliveryComplete` is called immediately.

At least once

QoS=1
Delivery is complete when an acknowledgment to the publication has been received from the queue manager. `MqttCallback.deliveryComplete` is called when the acknowledgment is received. The message might be delivered more than once before `MqttCallback.deliveryComplete` is called, if communications are slow or unreliable.

Exactly once

`QoS=2`
Delivery is complete when the client receives a completion message that the publication has been published to subscribers. `MqttCallback.deliveryComplete` is called as soon as the publication message is received. It does not wait for the completion message.

In rare circumstances, your client application might not return to the MQTT client from `MqttCallback.deliveryComplete` normally. You know that delivery has completed, because the `MqttCallback.deliveryComplete` was called. If the client restarts the same session, `MqttCallback.deliveryComplete` does not get called again.

Incomplete deliveries

If the delivery is not complete after the client session is disconnected you can connect the client again and complete the delivery. You can only complete the delivery of a message if the message was published in a session with the `MqttConnectionOptions` attribute set to `false`.

Create the client using the same client identifier and server address, and then connect, setting the `cleanSession` `MqttConnectionOptions` attribute to `false` again. If you set `cleanSession` to `true`, pending delivery tokens are thrown away.

You can check if there are any pending deliveries
by calling `MqttClient.getPendingDeliveryTokens`. You can call
`MqttClient.getPendingDeliveryTokens` before connecting the client.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Last will and testament publication

If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Create a topic for the last will and testament. You might create a topic such as `MQTTManagement/Connections/`*`server URI`*`/`*`client identifer`*`/Lost`.

Set up a "last will and testament" using the `MqttConnectionOptions.setWill(MqttTopic lastWillTopic, byte [] lastWillPayload, int lastWillQos, boolean lastWillRetained)` method.

Consider creating a time stamp in the `lastWillPayload` message. Include other client information that assists in identifying the client and the circumstances of the connection. Pass the `MqttConnectionOptions` object to the `MqttClient` constructor.

Set `lastWillQos` to 1 or 2, to make the message persistent in IBM MQ, and to guarantee delivery. To retain the last lost connection information, set the `lastWillRetained` to `true`.

The "last will and testament" publication is sent to subscribers if the connection ends unexpectedly. It is sent if the connection ends without the client calling the `MqttClient.disconnect` method.

To monitor connections, complement the "last will and testament" publication with other publications to record connections and programmed disconnections.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client

An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Message persistence in MQTT clients

Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

In MQTT, message persistence has two aspects; how the message is transferred, and whether it is queued in IBM MQ as a persistent message.

1. The MQTT client couples message persistence with quality of service. Depending on what quality of service you choose for a message, the message is made persistent. Message persistence is necessary to implement the required quality of service.

   If you specify "at most once", QoS=0, the client discards the message as soon as it is published. If there is any failure in the upstream processing of the message, the message is not sent again. Even if the client remains active the message is not sent again. The behavior of QoS=0 messages is the same as IBM MQ fast nonpersistent messages.

   If a message is published by a client with QoS of 1 or 2, it is made persistent. The message is stored locally, and only discarded from the client when it is no longer needed to guarantee "at least once", QoS=1, or "exactly once", QoS=2, delivery.
2. If a message is marked as QoS 1 or 2, it is queued in IBM MQ as a persistent message. If it is marked as QoS=0, then it is queued in IBM MQ as a nonpersistent message. In IBM MQ nonpersistent messages are transferred between queue managers "exactly once", unless the message channel has the NPMSPEED attribute set to FAST.

A persistent publication is stored on the client until it is received by a client application. For QoS=2, the publication is discarded from the client when the application callback returns control. For QoS=1 the application might receive the publication again, if a failure occurs. For QoS=0, the callback receives the publication no more than once. It might not receive the publication if there is a failure, or if the client is disconnected at the time of publication.

When you subscribe to a topic, you can reduce the QoS with which the subscriber receives messages to match its persistence capabilities. Publications that are created at a higher QoS are sent with the highest QoS that the subscriber requested.

Storing messages

The implementation of data storage on small devices varies a great deal. The model of temporarily saving persistent messages in storage that is managed by the MQTT client might be too slow, or demand too much storage. In mobile devices, the mobile operating system might provide a storage service that is ideal for MQTT messages.

To provide flexibility in meeting the constraints of small devices, the MQTT client has two persistence interfaces. The interfaces define the operations that are involved in storing persistent messages. The interfaces are described in the API documentation for the MQTT client for Java. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference. You can implement the interfaces to suit a device. The MQTT client that runs on Java SE has a default implementation of the interfaces that store persistent messages in the file system. It uses the `java.io` package.

Persistence classes

`MqttClientPersistence`
Pass an instance of your implementation of `MqttClientPersistence` to the MQTT client as a parameter of the `MqttClient` constructor. If you omit the `MqttClientPersistence` parameter from the `MqttClient` constructor, the MQTT client stores persistent messages using the class `MqttDefaultFilePersistence`.

`MqttPersistable`
`MqttClientPersistence` gets and puts `MqttPersistable` objects using a storage key. You must provide an implementation of `MqttPersistable` as well as the implementation of `MqttClientPersistence` if you are not using the `MqttDefaultFilePersistence`.

`MqttDefaultFilePersistence`

The MQTT client provides the `MqttDefaultFilePersistence` class. If you instantiate `MqttDefaultFilePersistence` in your client application, you can provide the directory to store persistent messages as a parameter of the `MqttDefaultFilePersistence` constructor.

Alternatively, the MQTT client can instantiate `MqttDefaultFilePersistence` and place files in the following default directory:

`client identifier` `-tcp` *`hostname portnumber`*

The following characters are removed from the directory name string:

`"\", "\\", "/", ":"` and `" "`

The path to the directory is the value of the system property `rcp.data`; If `rcp.data` is not set, the path is the value of the system property `usr.data`, where

- `rcp.data` is a property associated with installation of an OSGi or Eclipse Rich Client Platform (RCP).
- `usr.data` is the directory in which the Java command that started the application was launched.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication

If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Publications

Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

An `MqttMessage` has a byte array as its payload. Aim to keep messages as small as possible. The maximum length of message permitted by the MQTT protocol is 250 MB.

Typically, an MQTT client program uses `java.lang.String` or `java.lang.StringBuffer` to manipulate message contents. For convenience, the `MqttMessage` class has a `toString` method to convert its payload to a string. To create the byte array payload from a `java.lang.String` or `java.lang.StringBuffer`, use the `getBytes` method.

The `getBytes` method converts a string to the default character set for the platform. The default character set is generally `UTF-8`. MQTT publications that contain only text are usually encoded in `UTF-8`. Use the method `getBytes("UTF8")` to override the default character set.

In IBM MQ, an MQTT publication is received as a `jms-bytes` message. The message includes an `MQRFH2` folder containing an `<mqtt>`, and an `<mqps>` folder. The `<mqtt>` folder contains the `clientId`, `msgId`, and `qos`, but this content might change in the future.

An `MqttMessage` has three additional attributes: quality of service, whether it is retained, and whether it is a duplicate. The duplicate flag is only set if the quality of service is "at least once" or "exactly once". If the message was sent previously, and not acknowledged quickly enough by the MQTT client, the message is sent again, with the duplicate attribute set to `true`.

Publishing

To create a publication in an MQTT client application, create an `MqttMessage`. Set its payload, quality of service and whether it is retained, and call the `MqttTopic.publish(MqttMessage message)` method; `MqttDeliveryToken` is returned and the completion of the publication is asynchronous.

Alternatively, the MQTT client can create a temporary message object for you from the parameters on the `MqttTopic.publish(byte [] payload, int qos, boolean retained)` method when it creates a publication.

If the publication has an "at least once" or an "exactly once" quality of service, QoS=1 or QoS=2, the MQTT client calls the `MqttClientPersistence` interface. It calls `MqttClientPersistence` to store the message before returning a delivery token to the application.

The application can choose to block until the message is delivered to the server, using the `MqttDeliveryToken.waitForCompletion` method. Alternatively, the application can continue without blocking. If you want to check if publications are delivered, without blocking, register an instance of a callback class that implements `MqttCallback` with the MQTT client. The MQTT client calls the `MqttCallback.deliveryComplete` method as soon as the publication has been delivered. Depending on the quality of service, the delivery might be almost immediate for QoS=0, or it might take some time for QoS=2.

Use the `MqttDeliveryToken.isComplete` method to poll if delivery is complete. While the value of `MqttDeliveryToken.isComplete` is `false`, you can call `MqttDeliveryToken.getMessage` to get the message contents. If the result of calling `MqttDeliveryToken.isComplete` is `true`, the message has been discarded and calling `MqttDeliveryToken.getMessage` would throw a null pointer exception. There is no built-in synchronization between `MqttDeliveryToken.getMessage` and `MqttDeliveryToken.isComplete`.

If the client disconnects before receiving all the pending delivery tokens, a new instance of the client can query pending delivery tokens before connecting. Until the client connects, no new deliveries are completed, and it is safe to call `MqttDeliveryToken.getMessage`. Use the `MqttDeliveryToken.getMessage` method to find out which publications have not been delivered. Pending delivery tokens are discarded if you connect with `MqttConnectOptions.cleanSession` set to its default value, `true`.

Subscribing

A queue manager is responsible for creating publications to send to an MQTT subscriber. The queue manager checks if the topic filter in a subscription created by an MQTT client matches the topic string in a publication. The match can either be an exact match, or the match can include wildcards. Before the publication is forwarded to the subscriber by the queue manager, the queue manager checks the topic attributes associated with the publication. It follows the search procedure described in Subscribing using a topic string that contains wildcard characters to identify if an administrative topic object grants the user authority to subscribe.

When the MQTT client receives a publication with "at least once" quality of service, it calls the `MqttCallback.messageArrived` method to process the publication. If the quality of service of the publication is "exactly once", QoS=2, the MQTT client calls the `MqttClientPersistence` interface to store the message when it is received. It then calls `MqttCallback.messageArrived`.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Qualities of service provided by an MQTT client

An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

The quality of service of a publication is an attribute of `MqttMessage`. It is set by the method `MqttMessage.setQos`.

The method `MqttClient.subscribe` can reduce the quality of service applied to publications sent to a client on a topic. The quality of service of a publication forwarded to a subscriber might be different to the quality of service of the publication. The lower of the two values is used to forward a publication.

At most once
`QoS=0`

The message is delivered at most once, or it is not delivered at all. Its delivery across the network is not acknowledged.

The message is not stored. The message might be lost if the client is disconnected, or if the server fails.

`QoS=0` is the fastest mode of transfer. It is sometimes called "fire and forget".

The MQTT protocol does not require servers to forward publications at `QoS=0` to a client. If the client is disconnected at the time the server receives the publication, the publication might be discarded, depending on the server. The telemetry (MQXR) service does not discard messages sent with `QoS=0`. They are stored as nonpersistent messages, and are only discarded if the queue manager stops.

At least once
`QoS=1`

`QoS=1` is the default mode of transfer.

The message is always delivered at least once. If the sender does not receive an acknowledgment, the message is sent again with the DUP flag set until an acknowledgment is received. As a result, the receiver can be sent the same message multiple times, and might process it multiple times.

The message must be stored locally at the sender and the receiver until it is processed.

The message is deleted from the receiver after it has processed the message. If the receiver is a broker, the message is published to its subscribers. If the receiver is a client, the message is delivered to the subscriber application. After the message is deleted, the receiver sends an acknowledgment to the sender.

The message is deleted from the sender after it has received an acknowledgment from the receiver.

Exactly once

QoS=2

The message is always delivered exactly once.

The message must be stored locally at the sender and the receiver until it is processed.

QoS=2 is the safest, but slowest mode of transfer. It takes at least two pairs of transmissions between the sender and receiver before the message is deleted from the sender. The message can be processed at the receiver after the first transmission.

In the first pair of transmissions, the sender transmits the message and gets acknowledgment from the receiver that it has stored the message. If the sender does not receive an acknowledgment, the message is sent again with the DUP flag set until an acknowledgment is received.

In the second pair of transmissions, the sender tells the receiver that it can complete processing the message, "PUBREL". If the sender does not receive an acknowledgment of the "PUBREL" message, the "PUBREL" message is sent again until an acknowledgment is received. The sender deletes the message it saved when it receives the acknowledgment to the "PUBREL" message

The receiver can process the message in the first or second phases, provided that it does not reprocess the message. If the receiver is a broker, it publishes the message to subscribers. If the receiver is a client, it delivers the message to the subscriber application. The receiver sends a completion message back to the sender that it has finished processing the message.

Related concepts

Callbacks and synchronization in MQTT client applications

The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions

The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier

The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication

If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients

Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Retained publications and MQTT clients

A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Use the `MqttMessage.setRetained` method to specify whether a publication on a topic is retained.

When you create or update a retained publication, send the publication with a QoS of 1 or 2. If you send it with a QoS of 0, IBM MQ creates a nonpersistent retained publication. The publication is not retained if the queue manager stops.

If you publish a non-retained publication to a topic that has a retained publication, the retained publication is not affected. Current subscribers receive the new publication. New subscribers receive the retained publication first, then receive any new publications.

You can use a retained publication to record the latest value of a measurement. New subscribers to a topic immediately receive the most recent value of the measurement. If no new measurements are taken since the subscriber last subscribed to the publication topic, and if the subscriber subscribes again, the subscriber receives the most recent retained publication on the topic again.

To delete a retained publication, you have two options:

- Run the **`CLEAR TOPICSTR`** MQSC command.
- Create a zero-length retained publication. As specified in the MQTT 3.1.1 specification, if a zero-length retained message is published to a topic, any retained message for that topic is cleared.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Subscriptions

Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Create subscriptions using the `MqttClient.subscribe` methods, passing one or more topic filters and quality of service parameters. The quality of service parameter sets the maximum quality of service that the subscriber is prepared to use to receive a message. Messages sent to this client cannot be delivered with a higher quality of service. The quality of service is set to the lower of the original value when the message was published and the level specified for the subscription. The default quality of service for receiving messages is `QoS=1`, at least once.

The subscription request itself is sent with `QoS=1`.

Publications are received by a subscriber when the MQTT client calls the `MqttCallback.messageArrived` method. The `messageArrived` method also passes the topic string with which the message was published to the subscriber.

You can remove a subscription, or a set or subscriptions, using the `MqttClient.unsubscribe` methods.

An IBM MQ command can remove a subscription. List subscriptions using IBM MQ Explorer, or by using **runmqsc** or PCF commands. All MQTT client subscriptions are named. They are given a name of the form: *ClientIdentifier*:*Topic name*

If you use the default `MqttConnectOptions`, or set `MqttConnectOptions.cleanSession` to `true` before connecting the client, any old subscriptions for the client are removed when the client connects. Any new subscriptions the client makes during the session are removed when it disconnects.

If you set `MqttConnectOptions.cleanSession` to `false` before connecting, any subscriptions the client creates are added to all the subscriptions that existed for the client before it connected. All the subscriptions remain active when the client disconnects.

Another way of understanding the way the `cleanSession` attribute affects subscriptions is to think of it as a modal attribute. In its default mode, `cleanSession=true`, the client creates subscriptions and receives publications only within the scope of the session. In the alternative mode, `cleanSession=false`, subscriptions are durable. The client can connect and disconnect and its subscriptions remain active. When the client reconnects, it receives any undelivered publications. While it is connected, it can modify the set of subscriptions that are active on its behalf.

You must set the `cleanSession` mode before connecting; the mode lasts for the whole session. To change its setting, you must disconnect and reconnect the client. If you change modes from using `cleanSession=false` to `cleanSession=true`, all previous subscriptions for the client, and any publications that have not been received, are discarded.

Publications that match active subscriptions are sent to the client as soon as they are published. If the client is disconnected, they are sent to the client if it reconnects to the same server with the same client identifier and `MqttConnectOptions.cleanSession` set to `false`.

Subscriptions for a particular client are identified by the client identifier. You can reconnect the client from a different client device to the same server, and continue with the same subscriptions and receive undelivered publications.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the

default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Topic strings and topic filters in MQTT clients
Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Topic strings and topic filters in MQTT clients

Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM MQ.

Topics strings are used to send publications to subscribers. Create a topic string using the method, `MqttClient.getTopic(java.lang.String topicString)`.

Topic filters are used to subscribe to topics and receive publications. Topic filters can contain wildcards. With wildcards, you can subscribe to multiple topics. Create a topic filter by using a subscription method; for example, `MqttClient.subscribe(java.lang.String topicFilter)`.

Topic strings

The syntax of an IBM MQ topic string is described in Topic Strings. The syntax of MQTT topic strings is described in the `MqttClient` class in the API documentation for the MQTT client for Java. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

The syntax of each type of topic string is almost identical. There are four minor differences:

1. Topic strings sent to IBM MQ by MQTT clients must follow the convention for queue manager names.
2. The maximum lengths differ. IBM MQ topic strings are limited to 10,240 characters. An MQTT client can create topic strings of up to 65535 bytes.
3. A topic string created by an MQTT client cannot contain a null character.
4. In IBM Integration Bus, a null topic level, `'...//...'` is invalid. Null topic levels are supported by IBM MQ.

Unlike IBM MQ publish/subscribe, the `mqttv3` protocol does not have a concept of an administrative topic object. You cannot construct a topic string from a topic object and a topic string. However, a topic string is mapped to an administrative topic in IBM MQ. The access control associated with the administrative topic determines whether a publication is published to the topic, or discarded. The attributes that are applied to a publication when it is forwarded to subscribers, are influenced by the attributes of the administrative topic.

Topic filters

The syntax of an IBM MQ topic filter is described in Topic-based wildcard scheme. The syntax of the topic filters you can construct with an MQTT client are described in the `MqttClient` class in the API documentation for the MQTT client for Java. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

Related concepts

Callbacks and synchronization in MQTT client applications
The MQTT client programming model uses threads extensively. The threads decouple an MQTT client application, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Clean sessions
The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

Client identifier
The client identifier is a 23 byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. The identifier must contain only characters valid in a queue manager name. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

Delivery tokens

Last will and testament publication
If an MQTT client connection unexpectedly ends, you can configure MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Message persistence in MQTT clients
Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications sent to or from the client.

Publications
Publications are instances of `MqttMessage` that are associated with a topic string. MQTT clients can create publications to send to IBM MQ, and subscribe to topics on IBM MQ to receive publications.

Qualities of service provided by an MQTT client
An MQTT client provides three qualities of service for delivering publications to IBM MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM MQ to create a subscription, the request is sent with the "at least once" quality of service.

Retained publications and MQTT clients
A topic can have one, and only one, retained publication. If you create a subscription to a topic that has a retained publication, the publication is immediately forwarded to you.

Subscriptions
Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Developing Microsoft Windows Communication Foundation applications with IBM MQ

The Microsoft Windows Communication Foundation (WCF) custom channel for IBM MQ sends and receives messages between WCF clients and services.

Related concepts

“Introduction to the IBM MQ custom channel for WCF with .NET” on page 1219

The custom channel for IBM MQ is a transport channel using the Microsoft Windows Communication Foundation (WCF) unified programming model.

"Using IBM MQ custom channels for WCF" on page 1223
Overview of the information available for programmers using IBM MQ custom channels for Windows Communication Foundation (WCF).

"Using the WCF samples" on page 1242
The Windows Communication Foundation (WCF) samples provide some simple examples of how the IBM MQ custom channel can be used.

FFST: WCF XMS First Failure Support Technology

Related tasks

Tracing the WCF custom channel for IBM MQ

Troubleshooting WCF custom channel for IBM MQ problems

Introduction to the IBM MQ custom channel for WCF with .NET

The custom channel for IBM MQ is a transport channel using the Microsoft Windows Communication Foundation (WCF) unified programming model.

The Microsoft Windows Communication Foundation framework, introduced in Microsoft.NET 3, enables .NET applications and services to be developed independently from the transport and protocols used to connect them, enabling alternative transports or configurations to be used according to the environment that the service or application is deployed in.

Connections are managed at run time by WCF by building a channel stack containing the required combination of:

- Protocol elements: An optional set of elements where none, one, or more can be added to support protocols such as the WS-* standards.
- Message encoder: A mandatory element in the stack controlling the serializing of the message into its wire format.
- Transport channel: A mandatory element in the stack responsible for transporting the serialized message to its endpoint.

The custom channel for IBM MQ is a transport channel, and as such must be paired with a message encoder and optional protocols as required by the application using a WCF custom binding. In this way, applications which have been developed to use WCF can use the custom channel for IBM MQ to send and receive data in the same way as they use the built-in transports provided by Microsoft, enabling simple integration with the asynchronous, scalable, and reliable messaging functions of IBM MQ. For a full list of supported functions, see: "WCF Custom channel features and capabilities" on page 1223.

When and why do I use the IBM MQ custom channel for WCF?

You can use the IBM MQ custom channel to send and receive messages between WCF clients and services in the same way as the built-in transports provided by Microsoft, enabling applications to access the features of IBM MQ within the WCF unified programming model.

A typical usage pattern scenarios for the IBM MQ custom channel for WCF is as a non-SOAP interface for transmission of native IBM MQ messages.

Messages carried using the Non-SOAP/Non-JMS message (Pure MQMessage) format

When you use the IBM MQ custom channel for WCF as a non-SOAP interface for the transmission of native IBM MQ messages, the messages are carried by using the Non-SOAP/Non-JMS message (Pure MQMessage) format of IBM MQ.

WCF users are able to start the service, or in other words, service users are able to send a message to an IBM MQ queue by using MQMessages. Applications can get and set the MQMD fields and payload. When the message is available in IBM MQ queues, this message can be processed by any WCF service or non-WCF applications such as C or Java applications that are running on AIX, Linux, Windows, or z/OS.

Software requirements for the IBM MQ custom channel for WCF

This topic outlines the software requirements for the IBM MQ custom channel for WCF. The IBM MQ custom channel for WCF can only connect to IBM WebSphere MQ 7.0 or higher queue managers.

Runtime environment requirements

- Microsoft.NET Framework v4.7.2 or higher must be installed on the host machine.
- *Java and .NET Messaging and Web Services* is installed by default as part of the IBM MQ installer. This component installs the .NET assemblies needed for the custom channel into the Global Assembly Cache.

Note: If the Microsoft .NET Framework V4.7.2 or higher is not installed before you install IBM MQ, then the IBM MQ product installation continues without error, but the IBM MQ classes for .NET is not available. If the.NET Framework is installed after you install IBM MQ, then the IBM MQ.NET assemblies must be registered by running the *WMQInstallDir*`\bin\amqiRegisterdotNet.cmd` script, where *WMQInstallDir* is the directory where IBM MQ is installed. This script installs the required assemblies in the Global Assembly Cache (GAC). A set of `amqi*.log` files that record the actions that are taken are created in the %TEMP% directory. It is not necessary to rerun the `amqiRegisterdotNet.cmd` script if .NET is upgraded to V4.7.2 or higher from an earlier version, for example, from .NET V3.5.

Development environment requirements

- Microsoft Visual Studio 2015 or Windows Software Development Kit for .NET 4.7.2 or later.
- Microsoft.NET Framework V4.7.2 or higher must be installed on the host machine in order to build the sample solution files.

IBM MQ custom channel for WCF: What's installed?

The custom channel for IBM MQ is a transport channel using the Microsoft Windows Communication Foundation (WCF) unified programming model. The custom channel is installed by default as part of the installation.

IBM MQ custom channel for WCF

The custom channel and its dependencies are contained within the `Java and .NET Messaging and Web Services` component, which is installed by default. When upgrading IBM MQ from an earlier version thanIBM MQ 8.0, the update installs the IBM MQ custom channel for WCF by default if the `Java and .NET Messaging and Web Services` component was previously installed in an earlier installation.

The `.NET Messaging and Web Services` component contains the `IBM.XMS.WCF.dll` file, and the `IBM.WMQ.WCF.dll` file, and these files are the main custom channel assembly, which contains the WCF interface classes. These files are installed in the Global Assembly Cache (GAC) and are also available in the following directory: *MQ_INSTALLATION_PATH* `\bin` where *MQ_INSTALLATION_PATH* is the directory in which IBM MQ is installed.

The following table summarizes the key classes that are required for using the custom channel.

Table 189. Key classes required for using the custom channel

	SOAP/JMS interface (Existing)	**Non-SOAP/Non-JMS interface (From IBM MQ 8.0)**
Custom Channel Assembly	IBM.XMS.WCF.dll	IBM.WMQ.WCF.dll
Transport Binding Name	IBM.XMS.WCF.SoapJmsIbmTransport BindingElement	IBM.WMQ.WCF.WmqIbmTransportBindin gElement

Table 189. Key classes required for using the custom channel (continued)		
	SOAP/JMS interface (Existing)	**Non-SOAP/Non-JMS interface (From IBM MQ 8.0)**
Transport Binding Importer	IBM.XMS.WCF.SoapJmsIbmTransportBindingElementImporter	IBM.WMQ.WCF.WmqIbmTransportBindingElementImporter
Transport Binding Config	IBM.XMS.WCF.SoapJmsIbmTransportBindingElementConfig	IBM.WMQ.WCF.WmqIbmTransportBindingElementConfig
Samples(Oneway)	SimpleOneWay_Client, SimpleOneWay_Service	MQMessaging_OneWay_Client,MQMessaging_OneWay_Service
Samples(RequestReply)	SimpleRequestReply_Client, SimpleRequestReply_Service	MQMessaging_RequestReply_Client,MQMessaging_RequestReply_Service

IBM.WMQ.WCF.dll supports both SOAP/JMS and Non-SOAP/Non-JMS interfaces. New applications developed are recommended to use the IBM.WMQ.WCF assembly as it supports both interfaces.

Sending MQSTR formatted messages

If the request message is of type MQSTR, you can select to send the reply message in MQSTR format.

You must use an additional URI parameter **`replyMessageFormat`** to change the format of the reply message. The supported values are:

""

" " is the default value.

The reply message is in byte (MQMFT_NONE) format. For example:

```
"jms:/queue?
destination=SampleQ@QM1&connectionFactory=binding(server)connectQueueManager(QM1)
&initialContextFactory=com.ibm.mq.jms.Nojndi&replyDestination=SampleReplyQ&replyMessageForma
t= "
```

MQSTR

The reply message is in MQSTR (MQMFT_STRING) format. For example:

```
"jms:/queue?
destination=SampleQ@QM1&connectionFactory=binding(server)connectQueueManager(QM1)
&initialContextFactory=com.ibm.mq.jms.Nojndi&replyDestination=SampleReplyQ&replyMessageForma
t=MQSTR"
```

Notes:

1. The value for **`replyMessageFormat`** is case insensitive.
2. Using any value other than " " or *MQSTR*, causes an invalid parameter value exception.

IBM MQ custom channel samples

The samples provide some simple examples of how the IBM MQ custom channel for WCF can be used. The samples and their associated files are located in the *MQ_INSTALLATION_PATH* `\tools\dotnet\samples\cs\wcf` directory, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ. For more information on the IBM MQ custom channel samples, see "Using the WCF samples" on page 1242.

svcutil.exe.config

The `svcutil.exe.config` is an example of the configuration settings required to enable the Microsoft WCF `svcutil` client proxy generation tool to recognize the custom channel. The `svcutil.exe.config`

file is located in the *MQ_INSTALLATION_PATH* `\tools\wcf\docs\examples\` directory, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ. For more information on using the `svcutil.exe.config`, see "Generating a WCF client proxy and application configuration files using the svcutil tool with metadata from a running service" on page 1239.

WCF architecture

The IBM MQ custom channel for WCF is integrated on top of the `IBM Message Service Client for .NET (XMS .NET)` API.

SOAP/JMS interface

The WCF architecture is as shown in the following diagram:

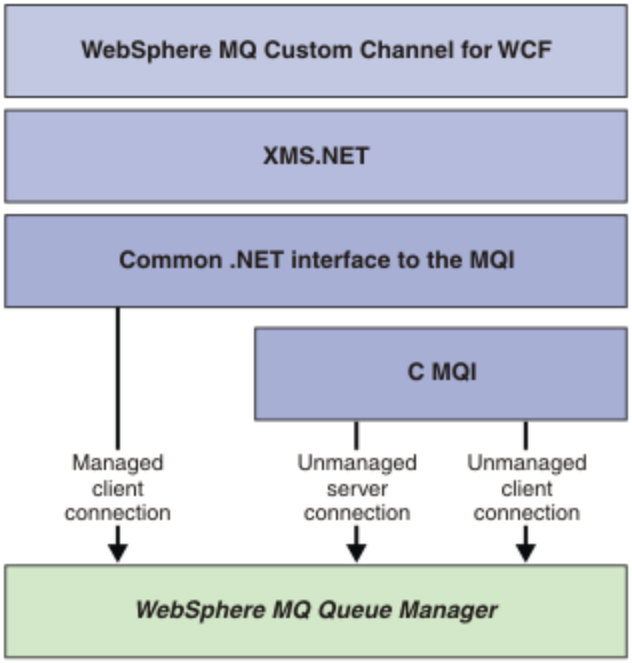


Figure 149. WCF architecture for the SOAP/JMS interface

All the required components are installed by default with the product installation.

The three connections are:

- Managed client connections
- Unmanaged server connections
- Unmanaged client connections

For more information about these connections, see "WCF Connection options" on page 1229.

Non-SOAP/Non-JMS interface

The IBM MQ custom channel for WCF supports both the SOAP/JMS interface (available from IBM WebSphere MQ 7.0.1) and the Non-SOAP/Non-JMS interface.

The WCF architecture is as shown in the following diagram:

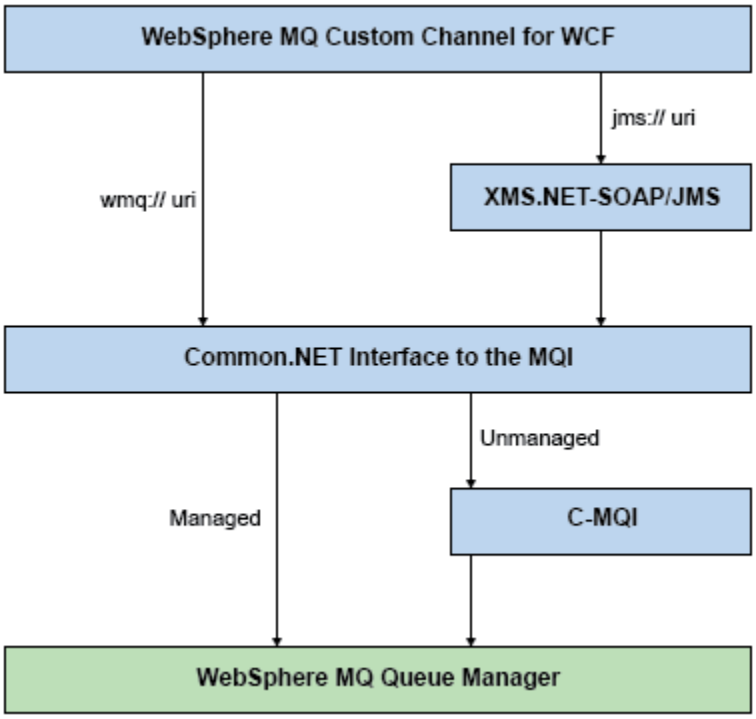


Figure 150. WCF architecture for the Non-SOAP/Non-JMS interface

Using IBM MQ custom channels for WCF

Overview of the information available for programmers using IBM MQ custom channels for Windows Communication Foundation (WCF).

The Microsoft Windows Communication Foundation underpins the web services and messaging support in the Microsoft.NET Framework 3. IBM MQ can be used as a custom channel within WCF in the .NET Framework 3 in the same manner as the built-in channels offered by Microsoft.

Messages transported across the custom channel are formatted according to the SOAP over JMS implementation of IBM MQ. Applications can then communicate with services hosted by WCF or by the WebSphere SOAP over JMS service infrastructure.

WCF Custom channel features and capabilities

Use the following topics for information regarding WCF custom channel features and capabilities.

WCF custom channel shapes

Overview of the custom channel shapes that IBM MQ can be used as within the Microsoft Windows Communication Foundation (WCF) custom channels.

The IBM MQ custom channel for WCF supports two channel shapes:

- One-way
- Request-reply

WCF automatically selects the channel shape according to the service contract being hosted.

Contracts that include methods that only use the **IsOneWay** parameter are serviced by the one-way channel shape, for example:

```
[OperationContract(IsOneWay = true)]
void printString(String text);
```

Contracts that include either a mixture of one-way and request-reply methods, or all request-reply methods, are serviced by the request-reply channel shape. For example:

```
[OperationContract]
int subtract(int a, int b);

[OperationContract(IsOneWay = true)]
void printString(string text);
```

Note: When mixing one-way and request-reply methods in the same contract, you must ensure that the behavior is as intended, especially when working within a mixed environment because one-way methods wait until they receive a null reply from the service.

One-way channel

The IBM MQ one-way custom channel for WCF is used, for example, to send messages from a WCF client using a one-way channel shape. The channel can send messages in one direction only, for example; from a client queue manager to a queue on a WCF service.

Request-reply channel

The IBM MQ request-reply custom channel for WCF is used, for example, to send messages in two directions asynchronously; The same client instance must be used for asynchronous messaging. The channel can send messages in one direction, for example; from a client queue manager to a queue on a WCF service, and then send a reply message from the WCF to a queue on the client queue manager.

WCF URI parameter names and values

URI parameter names and values for the SOAP/JMS interface and Non-SOAP/Non JMS interface.

SOAP/JMS interface

connectionFactory
The connectionFactory parameter is required.

initialContextFactory
The initialContextFactory parameter is required and must be set to "com.ibm.mq.jms.Nojndi" for compatibility with WebSphere Application Server and other products.

Non-SOAP/Non JMS interface

The URI format is as for the MA93 specifications. See SupportPac - MA93 for further details of the IBM MQ IRI specifications.

IBM MQ URI syntax

```
wmq-iri = "wmq:" [ "//" connection-name ] "/" wmq-dest ["?" parm *("&" parm)]
connection-name = tcp-connection-name / other-connection-name
tcp-connection-name = ihost [ ":" port ]
other-connection-name = 1*(iunreserved / pct-encoded)
wmq-dest = queue-dest / topic-dest
queue-dest = "msg/queue/" wmq-queue ["@" wmq-qmgr]
wmq-queue = wmq-name
wmq-qmgr = wmq-name
wmq-name = 1*48( wmq-char )
topic-dest = "msg/topic/" wmq-topic
wmq-topic = segment *( "/" segment )
```

IBM MQ IRI example
The following example IRI tells a service requester that it can use a IBM MQ TCP client-binding connection to a machine called example.com on port 1414 and put persistent request messages to

a queue called SampleQ on queue manager QM1. The IRI specifies that the service provider will put replies to a queue called SampleReplyQ.

```
1)wmq://example.com:1414/msg/queue/SampleQ@QM1?
ReplyTo=SampleReplyQ&persistence=MQPER_NOT_PERSISTENT
2)wmq://localhost:1414/msg/queue/Q1?
connectQueueManager=QM1&replyTo=Q2&connectionmode=managed
```

For TLS enabled connections

To make Secured (TLS) connections using the WCF Client/Service, set following properties with appropriate values in the URI. All the properties that are prefixed with "*" are mandatory to make a secured connection.

- **sslKeyRepository**: *SYSTEM or *USER
- * **sslCipherSpec**: a valid CipherSpec, for example TLS_RSA_WITH_AES_128_CBC_SHA256.
- **sslCertRevocationCheck**: true or false.
- **sslKeyResetCount**: a value greater than 32kb.
- **sslPeerName:** the distinguished name of the server certificate

For example:

```
"wmq://localhost:1414/msg/queue/SampleQ?
connectQueueManager=QM1&sslkeyrepository=*SYSTEM&sslcipherspec=
TLS_RSA_WITH_AES_128_CBC_SHA&sslcertrevocationcheck=true&"sslpe
ername=" + "" + "CN=ibmwebspheremqqmm&sslkeyresetcount=45000"
```

WCF custom channel assured delivery

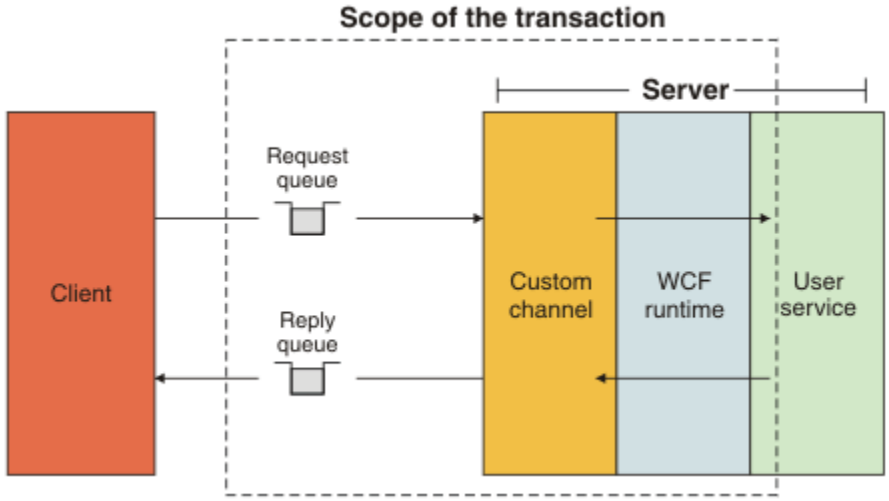
Assured Delivery guarantees that a service request or reply is actioned and not lost.

A request message is received and any reply message is sent under a local transaction sync point, which can be rolled back in the case of runtime failure. Examples of these failures are: An unhandled exception thrown by the service, failure to dispatch the message to the service, or failure to deliver the reply message.

`AssuredDelivery` is the assured delivery attribute which can be specified on a service contract to guarantee that any request messages received by a service, and any reply message sent from a service, is not lost in the event of a runtime failure.

To ensure that messages are also preserved in the event of system failure or power outage, messages must be sent as persistent. To use persistent messages the client application must have this option specified on its endpoint URI.

Distributed transactions are not supported, and the scope of the transaction does not extend beyond the request and reply message processing performed by IBM MQ. Any work performed within the service might get rerun as a result of a failure which causes the message to be received again. The following diagram shows the scope of the transaction:



Assured delivery is enabled by applying the `AssuredDelivery` attribute to the service class as shown in the following example:

```
[AssuredDelivery]
class TestCalculatorService : IWMQSampleCalculatorContract
{
    public int add(int a, int b)
    {
        int ans = a + b;
        return ans;
    }
}
```

When using the `AssuredDelivery` attribute, you must be aware of the following points:

- When a channel determines that a failure is likely to recur if a message was rolled-back and received again, the message is treated as a poison message and is not returned to the request queue for reprocessing. For example: If the received message is not correctly formatted or cannot be dispatched to a service. Unhandled exceptions thrown from a service operation are always resent until the message has been redelivered the maximum number of times specified by the backout threshold property of the request queue. For more information, see: “WCF custom channel poison messages” on page 1227
- The channel performs the reading, processing, and replying of each request message as an atomic operation using a single thread of execution to enforce transactional integrity. To enable service operations to run concurrently, the channel enables WCF to create multiple instances of the channel. The number of channel instances available for processing requests is controlled by the binding property `MaxConcurrentCalls`. For more information, see: “WCF binding configuration options” on page 1235
- The assured delivery function uses both the IOperationInvoker and the IErrorHandler WCF extensibility points. If these extensibility points are used externally by an application, the application must ensure that any previously registered extensibility points are called. Failure to do so for IErrorHandler can result in errors going unreported. Failure to do so for IOperationInvoker can cause WCF to stop responding.

WCF custom channel security

The IBM MQ custom channel for WCF supports the use of TLS only for unmanaged client connections to the queue manager.

Specify TLS using an entry in the client channel definition table (CCDT). For more information about CCDTs, see Client channel definition table.

WCF client channel definition tables (CCDT)

The IBM MQ custom channel for WCF supports the use of client channel definition tables (CCDT) to configure the connection information for client connections.

CCDTs are controlled through these two environment variables:

- *MQCHLLIB* specifies the directory where the table is located.
- *MQCHLTAB* specifies the file name of the table.

If these environment variables are defined, then they take priority over any client connection details specified in the URI.

For more information about client channel definition tables, see: Client channel definition table.

WCF custom channel poison messages

When a service fails to process a request message, or fails to deliver a reply message to a reply queue, then the message is treated as a poison message.

Poison request messages

If a request message cannot be processed, then it is treated as a poison message. This action prevents the service from receiving the same unprocessable message again. For an unprocessable request message to be treated as a poison message, one of the following situations must be true:

- The messages backout count exceeded the backout threshold specified on the request queue, which only occurs if assured delivery was specified for the service. For more information about assured delivery, see: "WCF custom channel assured delivery" on page 1225
- The message was not formatted correctly and could not be interpreted as a SOAP over JMS message.

Poison reply messages

If a service fails to deliver a reply message to the reply queue, then the reply message is treated as a poison message. For reply messages, this action enables the reply messages to be retrieved later to aid problem determination.

Poison message handling

The action taken for a poison message depends on the queue manager configuration and the values set in the report options of the message. For SOAP over JMS, the following report options are set on request messages by default and are not configurable:

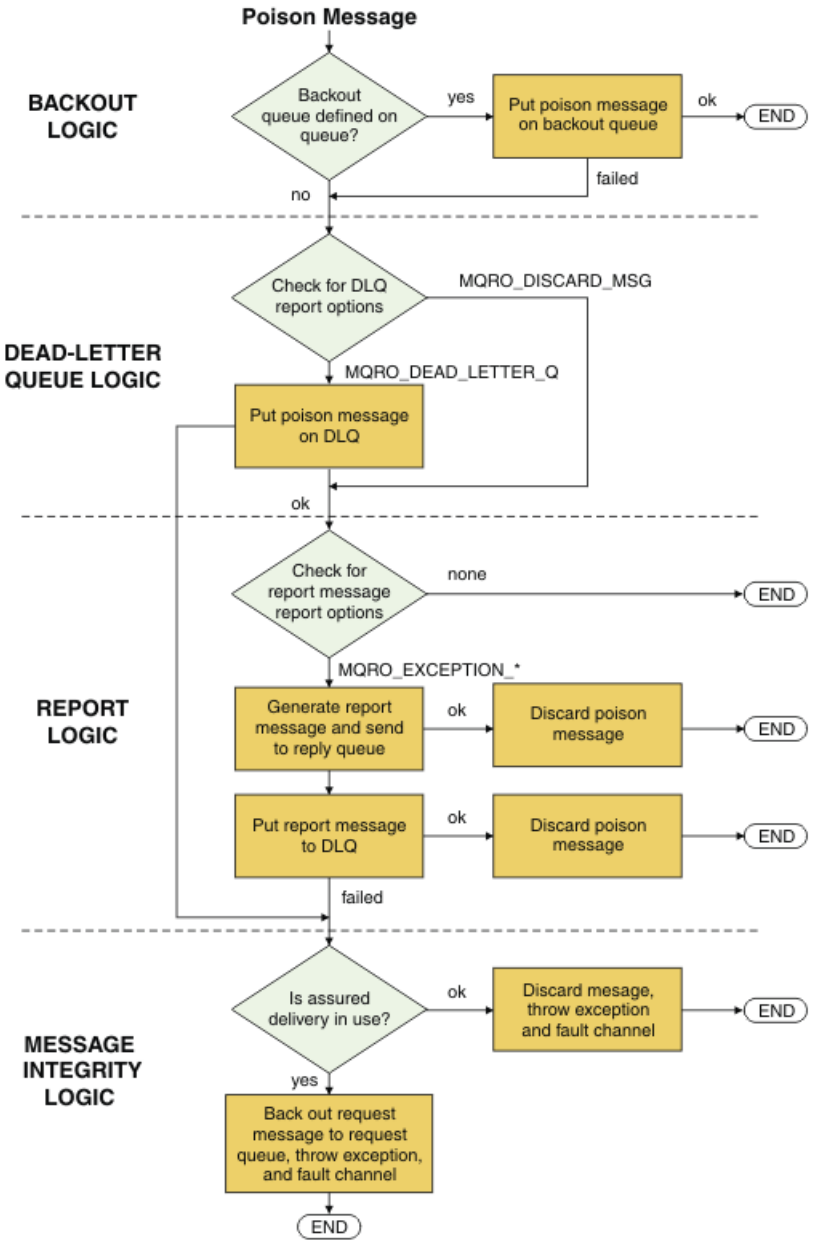
- MQRO_EXCEPTION_WITH_FULL_DATA
- MQRO_EXPIRATION_WITH_FULL_DATA
- MQRO_DISCARD_MSG

For SOAP over JMS, the following report option is set on reply messages by default and is not configurable:

- MQRO_DEAD_LETTER_Q

If messages come from a non-WCF source, then refer to the documentation for that source.

The following diagram shows the possible actions and the steps taken if poison message handling fails:



IBM MQ message capabilities for WCF applications

Non-SOAP/Non-JMS (that is, IBM MQ) message capabilities for WCF applications.

For the Non-SOAP/Non-JMS interface, the IBM MQ message capabilities for WCF applications are as follows:

- WCF applications can send and receive the base IBM MQ messages which can be processed by any IBM MQ application.
- WCF applications have full control to update the MQMD and payload.
- The WCF client can send IBM MQ messages that can be consumed by any IBM MQ clients, for example C, Java, JMS, and .NET clients.

The WCF for Non-SOAP/Non-JMS interface must use the following classes for setting the message payload and MQMD for the message:

- WmqStringMessage for a payload of type String
- WmqBytesMessage for a payload of type Bytes
- WmqXmlMessage for a payload of type XML

To set the payload of the message, use the **Data** property for the WmqStringMessage, WmqBytesMessage or WmqXmlMessage class, depending on the payload type. For example, use the following code to set a payload of type String:

```
WmqStringMessage strMsg = new WmqStringMessage();
//Setting the Message PayLoad
strMsg.Data = "Hello World";
//MQMD property
strMsg.Format = WmqMessageFormat.MQFMT_STRING;
```

WCF Connection options

There are three modes of connecting an IBM MQ custom channel for WCF to a queue manager. Consider which type of connection best suits your requirements.

For more information about connection options, see: “Connection differences” on page 559

For more information about WCF architecture, see: “WCF architecture” on page 1222

Unmanaged client connection

A connection made in this mode connects as an IBM MQ client to an IBM MQ server running either on the local machine or on a remote machine.

To use the IBM MQ custom channel for WCF as an IBM MQ client, you can install it, with the IBM MQ MQI client, either on the IBM MQ server, or on a separate machine.

Unmanaged server connection

When used in server bindings mode, the IBM MQ custom channel for WCF uses the queue manager API, rather than communicating through a network. Using bindings connections provides better performance for IBM MQ applications than using network connections.

To use the bindings connection, you must install the IBM MQ custom channel for WCF on the IBM MQ server.

Managed client connection

A connection made in this mode connects as an IBM MQ client to an IBM MQ server running either on the local machine or on a remote machine.

The IBM MQ custom channel classes for .NET 3 connecting in this mode remain in .NET managed code and make no calls to native services. For more information about managed code, see Microsoft documentation.

There are a number of limitations to using the managed client. For more information about these limitations, see “Managed client connections” on page 559.

Creating and configuring the IBM MQ custom channel for WCF

The IBM MQ custom channels for WCF work in the same manner as transport WCF channels offered by Microsoft. The IBM MQ custom channel for WCF can be created in one of two ways.

About this task

The IBM MQ custom channel integrates with WCF as a WCF transport channel, and as such must be paired with a message encoder and optional protocol channels, so it can create a complete channel stack that can be used by an application. Two elements are required for a complete channel stack to be created successfully:

1. A binding definition: Specifies which elements are required to build the applications channel stack, including transport channel, message encoder, and any protocols, plus any general configuration

settings. For the custom channel, the binding definition must be created in the form of a WCF custom binding.

2. An endpoint definition: Links the service contract with the binding definition, and also provides the actual connection URI which describes where the application can connect. For the custom channel, the URI is in the form of a SOAP over JMS URI.

These definitions can be created in one of two different ways:

- Administratively; The definitions are created by providing the details in an application configuration file (for example: `app.config`).
- Programmatically; The definitions are created directly from the application code.

The decision over which method to use to create the definitions must be based on the requirements of the application as follows:

- The Administrative method for configuration provides the flexibility to alter the details of the service and client post-deployment without rebuilding the application.
- The Programmatic method for configuration provides greater protection from configuration errors, and the ability to dynamically generate a configuration at run time.

Creating a WCF custom channel administratively by supplying binding and endpoint information in an application configuration file

The IBM MQ custom channel for WCF is a transport level WCF channel. An endpoint and binding must be defined to use the custom channel, and these definitions can be done by supplying the binding and endpoint information in an application configuration file.

To configure and use the IBM MQ custom channel for WCF, which is a transport level WCF channel, a binding and an endpoint definition must be defined. The binding holds the configuration information for the channel, and the endpoint definition holds the connection details. These definitions can be created in two ways:

- Programmatically directly from the application code, as described here: "Creating a WCF custom channel by suppling binding and endpoint information programmatically" on page 1232
- Administratively, by providing the details in an application configuration file, as described in the following procedure.

The client or service application configuration file is commonly named *yourappname*`.exe.config` where *yourappname* is the name of your application. The application configuration file is most easily modified by using the Microsoft service configuration editor tool called `SvcConfigEditor.exe` in the following way:

- Start the `SvcConfigEditor.exe` configuration editor tool. The default installation location for the tool is: *`Drive:`*`\Program Files\Microsoft SDKs\Windows\v6.0\Bin\SvcConfigEditor.exe` where *Drive:* is the name of the installation drive.

Step 1: Add a binding element extension to enable WCF to locate the custom channel

1. Right-click **Advanced** > **Extension** > **binding element** to open the menu, and select **New**
2. Complete the fields as shown in this table:

Table 190. New binding element fields

Field	Value
Name	`IBM.XMS.WCF.SoapJmsIbmTransportChannel`
Type	Navigate to `IBM.XMS.WCF.dll` in the Global Assembly Cache (GAC) and select `IBM.XMS.WCFSoapJmsIbmTransportBindingElementConfig`

Step 2: Create a custom binding definition which pairs the custom channel with a WCF message encoder

1. Right-click **Bindings** to open the menu, and select **New Binding Configuration**
2. Complete the fields as shown in this table:

Table 191. New binding configuration fields

Field	Value
Name	`CustomBinding_WMQ`
BindingElement 1	`textMessageEncoding (MessageVersion: Soap11)`
BindingElement 2	`IBM.XMS.WCF.SoapJmsIbmTransportChannel`

Step 3: Specify the binding properties

1. Select the *IBM.XMS.WCF.SoapJmsIbmTransportChannel* transport binding from the binding you created in: "Step 2: Create a custom binding definition which pairs the custom channel with a WCF message encoder" on page 1231
2. Make any required changes to the default values of the properties as described in: "WCF binding configuration options" on page 1235

Step 4: Create an endpoint definition

Create an endpoint definition which references the custom binding you created in: "Step 2: Create a custom binding definition which pairs the custom channel with a WCF message encoder" on page 1231 and provides the connection details of the service. The way this information is specified is dependent on whether the definition is for a client application or a service application.

For a client application, add an endpoint definition to the client section as follows:

1. Right-click **Client** > **Endpoints** to open the menu, and select **New Client Endpoint**
2. Complete the fields as shown in this table:

Table 192. New client endpoint fields

Field	Value
Name	`Endpoint_WMQ`
Address	*The SOAP/JMS URI describing the WMQ connection details required to access the service.* For further details see: "IBM MQ custom channel for WCF endpoint URI address format" on page 1234
Binding	`customBinding`
BindingConfiguration	`CustomBinding_WMQ`
Contract	*The name of your service contract interface*

For a service application, add a service definition to the services section as follows:

1. Right-click **Services** to open the menu, and select **New Service**, then select the service class to be hosted.
2. Add an endpoint definition to the **Endpoints** section for your new service, and complete the fields as shown in this table:

Table 193. New service endpoint fields	
Field	**Value**
Name	`Endpoint_WMQ`
Address	*The SOAP/JMS URI describing the WMQ connection details required to access the service.* For further details see: "IBM MQ custom channel for WCF endpoint URI address format" on page 1234
Binding	`customBinding`
BindingConfiguration	`CustomBinding_WMQ`
Contract	*The name of your service implementation class*

Creating a WCF custom channel by suppling binding and endpoint information programmatically

The IBM MQ custom channel for WCF is a transport level WCF channel. An endpoint and binding must be defined to use the custom channel, and these definitions can be done programmatically directly from the application code.

To configure and use the IBM MQ custom channel for WCF, which is a transport level WCF channel, a binding and an endpoint definition must be defined. The binding holds the configuration information for the channel, and the endpoint definition holds the connection details. For more information see "Using the WCF samples" on page 1242.

These definitions can be created in two ways:

- Administratively, by providing the details in an application configuration file, as described in "Creating a WCF custom channel administratively by supplying binding and endpoint information in an application configuration file" on page 1230.
- Programmatically directly from the application code, as described in the following subtopics.

Defining binding and endpoint information programmatically: SOAP/JMS interface

For the SOAP/JMS interface, you can define an endpoint and binding programmatically directly from the application code.

About this task

To supply binding and endpoint information programmatically, add the required code to your application by completing the following steps.

Procedure

1. Create an instance of the transport binding element of the channel by adding the following code to your application:

```
SoapJmsIbmTransportBindingElement transportBindingElement = new
SoapJmsIbmTransportBindingElement();
```

2. Set any required binding properties, for example, by adding the following code to your application to set the `ClientConnectionMode`:

```
transportBindingElement.ClientConnectionMode = XmsWCFBindingProperty.AS_URI;
```

3. Create a custom binding that pairs the transport channel with a message encoder by adding the following code to your application:

```
Binding binding = new CustomBinding(new TextMessageEncodingBindingElement(),
transportBindingElement);
```

4. Create the SOAP/JMS URI.

   The SOAP/JMS URI that describes the IBM MQ connection details required to access the service, must be provided as the endpoint address. The address that you specify depends on whether the channel is being used for a service application or a client application.

   - For client applications, the SOAP/JMS URI must be created as an `EndpointAddress` as follows:

     ```
     EndpointAddress address = new EndpointAddress("jms:/queue?
     destination=SampleQ@QM1&connectionFactory
     =connectQueueManager(QM1)&initialContextFactory=com.ibm.mq.jms.Nojndi");
     ```

   - For service applications, the SOAP/JMS URI must be created as a URI as follows:

     ```
     Uri address = new Uri("jms:/queue?destination=SampleQ@QM1&connectionFactory=
     connectQueueManager(QM1)&initialContextFactory=com.ibm.mq.jms.Nojndi");
     ```

     For more information about endpoint addresses, see “IBM MQ custom channel for WCF endpoint URI address format” on page 1234.

Defining binding and endpoint information programmatically: Non-SOAP/Non-JMS interface
For the Non-SOAP/Non-JMS interface, you can define an endpoint and binding programmatically directly from the application code.

About this task

To supply binding and endpoint information programmatically, add the required code to your application by completing the following steps.

Procedure

1. Create a WmqBinding by adding the following code to your application:

   ```
   WmqBinding binding = new WmqBinding();
   ```

   This code creates a binding that pairs the WmqMsgEncodingElement and WmqIbmTransportBindingElement required for the Non-SOAP/Non-JMS interface.

2. Provide the wmq:// URI that describes the IBM MQ connection details required to access the service.

   The way in which you provide the wmq:// URI depends on whether the channel is being used for a service application or a client application.

   - For client applications, the wmq:// URI must be created as an EndpointAddress as follows:

     ```
     EndpointAddress address = new EndpointAddress
     ("wmq://localhost:1414/msg/queue/Q1?connectQueueManager=QM1&replyTo=Q2");
     ```

   - For service applications, the wmq:// URI must be created as a URI as follows:

     ```
     Uri sampleAddress = new Uri(
     "wmq://localhost:1414/msg/queue/Q1?connectQueueManager=QM1&replyTo=Q2");
     ```

IBM MQ custom channel for WCF endpoint URI address format

A web service is specified using a Universal Resource Identifier (URI) that provides location and connection details. The URI format depends on whether you are using the SOAP/JMS interface or the Non-SOAP/Non-JMS interface.

SOAP/JMS interface

The URI format that is supported in the IBM MQ transport for SOAP permits a comprehensive degree of control over SOAP/ IBM MQ -specific parameters and options when accessing target services. This format is compatible with WebSphere Application Server and with CICS, facilitating the integration of IBM MQ with both those products.

The URI syntax is as follows:

```
jms:/queue? name=value&name=value...
```

where name is a parameter name and *value* is an appropriate value, and the name = *value* element can be repeated any number of times with the second and subsequent occurrences being preceded by an ampersand (&).

Parameter names are case-sensitive, as are names of IBM MQ objects. If any parameter is specified more than once, the final occurrence of the parameter takes effect meaning client applications can override parameter values by appending to the URI. If any additional unrecognized parameters are included, they are ignored.

If you store a URI in an XML string, you must represent the ampersand character as "&". Similarly, if a URI is coded in a script, take care to escape characters such as **&** which would otherwise be interpreted by the shell.

This is an example of a simple URI for an Axis service:

```
jms:/queue?destination=myQ&connectionFactory=()
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

Here is an example of a simple URI for a .NET service:

```
jms:/queue?destination=myQ&connectionFactory=()&targetService=MyService.asmx
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

Only the required parameters are supplied (`targetService` is required for .NET services only), and `connectionFactory` is given no options.

In this Axis example, `connectionFactory` contains a number of options:

```
jms:/queue?destination=myQ@myRQM&connectionFactory=connectQueueManager(myconnQM)
binding(client)clientChannel(myChannel)clientConnection(myConnection)
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

In this Axis example, the `sslPeerName` option of `connectionFactory` has also been specified. The value of sslPeerName itself contains name value pairs and significant embedded blanks:

```
jms:/queue?destination=myQ@myRQM&connectionFactory=connectQueueManager(myconnQM)
binding(client)clientChannel(myChannel)clientConnection(myConnection)
sslPeerName(CN=MQ Test 1,O=IBM,S=Hampshire,C=GB)
&initialContextFactory=com.ibm.mq.jms.Nojndi
```

NON-SOAP/Non-JMS interface

The URI format for the NON-SOAP/Non-JMS interface permits a comprehensive degree of control over IBM MQ -specific parameters and options when accessing target services.

The URI syntax is as follows:

```
wmq://example.com:1415/msg/queue/INS.QUOTE.REQUEST@MOTOR.INS ?ReplyTo=msg/queue/
INS.QUOTE.REPLY@BRANCH452&persistence=MQPER_NOT_PERSISTENT
```

This IRI tells a service requester that it can use an IBM MQ TCP client-binding connection to a machine called example.com on port 1415 and put persistent request messages to a queue called INS.QUOTE.REQUEST on queue manager MOTOR.INS. The IRI specifies that the service provider puts replies to a queue called INS.QUOTE.REPLY on queue manager BRANCH452. The URI format is as specified for SupportPac MA93. See SupportPac MA93: IBM MQ - Service Definition for more details about the IBM MQ IRI specifications.

WCF binding configuration options

There are two ways of applying configuration options to the custom channels binding information. You either set the properties administratively, or set them programmatically.

The binding configuration options can be set in one of two different ways:

1. Administratively: The binding property settings must be specified in the transport section of the custom binding definition in the applications configuration file, for example: `app.config`.
2. Programmatically: The application code must be modified to specify the property during initialization of the custom binding.

Setting the binding properties administratively

The binding property settings can be specified in the application configuration file, for example: `app.config`. The configuration file is generated by **`svcutil`**, as shown in the following examples.

SOAP/JMS interface

```
<customBinding>
...
      <IBM.XMS.WCF.SoapJmsIbmTransportChannel maxBufferPoolSize="524288"
      maxMessageSize="4000000" clientConnectionMode="0" maxConcurrentCalls="16"/>
...
</customBinding>
```

Non-SOAP/Non-JMS interface

```
<customBinding>
      <IBM.WMQ.WCF.WmqMsgEncodingElement/>
      <IBM.WMQ.WCF.WmqIbmTransportChannel maxBufferPoolSize="524288"
      maxMessageSize="65536" clientConnectionMode="managedclient"/>
</customBinding>
```

Setting the binding properties programmatically

To add a WCF binding property to specify the client connection mode, you must modify the service code to specify the property during initialization of the custom binding.

Use the following example to specify unmanaged client connection mode:

```
SoapJmsIbmTransportBindingElement
transportBindingElement = new SoapJmsIbmTransportBindingElement();
transportBindingElement.ClientConnectionMode = XmsWCFBindingProperty.CLIENT_UNMANAGED;

Binding sampleBinding = new CustomBinding(new TextMessageEncodingBindingElement(),
                                                    transportBindingElement);
```

WCF binding properties

Table 194. Values of binding properties when setting administratively or programmatically

Property name	Client or Service application	Administrative value	Programmatic value	Description
maxBufferPoolSize	Both	0 to 64 bit signed integer	0 to 64 bit signed integer	Specifies the maximum size of the memory that can be used to store WCF message buffers for an instance of the channel.
maxMessageSize	Both	1 to 32 bit signed integer	1 to 32 bit signed integer	Specifies the maximum memory that can be used for an individual WCF message.
clientConnectionMode	Both	0 (Default value) 1	AS_URI (Default value) CLIENT_UNMANAGED	Specifies the client connection mode of the transport channel. 0 means that the client connection mode is as specified in the URI. Only used if the client connection is used. Specifies that the client connection mode is as specified in the URI. 0 is the default value if no client connection mode is set. 1 means that the client connection mode is an unmanaged client. Only used if the client connection is used.
MaxConcurrentCalls	Client	The range is 0 - 2 147 483 647 16 is the default value	The range is 0 - 2 147 483 647 16 is the default value	This property defines the maximum number of concurrent operations that can take place on an individual client proxy at any one time. If more operations are started, they are queued until an in-progress operation either completes or times out. This setting can be used to control the maximum threads and resources which can be consumed by an individual proxy. 0 removes this limit, enabling all operations to be attempted concurrently.

Table 194. Values of binding properties when setting administratively or programmatically (continued)				
Property name	**Client or Service application**	**Administrative value**	**Programmatic value**	**Description**
MaxConcurrentCalls	Service	The range is 1 - 2 147 483 647 16 is the default value	The range is 1 - 2 147 483 647 16 is the default value	This property is only used if the assured delivery feature is enabled (For more information about assured delivery, see "WCF custom channel assured delivery" on page 1225). It specifies the maximum number of concurrent operations that can be in progress at the same time for the given endpoint. Care is needed when changing this setting. Each concurrent operation requires additional resources, in particular a new instance of the custom channel and the associated threads from the thread pool to action the requests. Over-allocating can be counter productive and affect performance severely. Appropriate configuration of the thread pool must be made to support this property.

Building and hosting services for WCF

Overview of Microsoft Windows Communication Foundation (WCF) services explaining how to create and configure WCF services.

The IBM MQ custom channel for WCF and the WCF services which use it, can be hosted by the following methods:

- Self-hosting
- Windows Service

The IBM MQ custom channel for WCF cannot be hosted in Windows Process Activation Service.

The following topics provide some simple self-hosting examples to demonstrate the steps involved. The Microsoft WCF online documentation, which contains further information and the latest details, can be found on the Microsoft MSDN website at https://msdn.microsoft.com.

Building WCF service applications using method 1: Self-hosting administratively using an application configuration file

Having created an application configuration file, open an instance of the service and add the specified code to your application.

Before you begin

Create or edit an application configuration file for the service, as described in: "Creating a WCF custom channel administratively by supplying binding and endpoint information in an application configuration file" on page 1230

About this task

1. Instantiate and open an instance of the service in the service host. The service type must be the same as the service type specified in the service configuration file.
2. Add the following code to your application:

```
ServiceHost service = new ServiceHost(typeof(MyService));
service.Open();
...
service.Close();
```

Building WCF service applications using method 2: Self-hosting programmatically directly from the application

Add the binding properties, create the service host with an instance of the required service class and open the service.

Before you begin

1. Add a reference to the custom channel `IBM.XMS.WCF.dll` file to the project. The `IBM.XMS.WCF.dll` is in the *`WMQInstallDir`*`\bin` where *WMQInstallDir* is the directory that IBM MQ is installed in.
2. Add a *using* statement to the `IBM.XMS.WCF` namespace, for example: `using IBM.XMS.WCF`
3. Create an instance of the channels binding element and endpoint as described in: "Creating a WCF custom channel by suppling binding and endpoint information programmatically" on page 1232

About this task

If changes to the binding properties of the channel are required, then complete the following steps:

1. Add the binding properties to `transportBindingElement` as shown in the following example:

```
SoapJmsIbmTransportBindingElement transportBindingElement = new
SoapJmsIbmTransportBindingElement();
Binding binding = new CustomBinding(new TextMessageEncodingBindingElement(),
transportBindingElement);
Uri address = new Uri("jms:/queue?destination=SampleQ@QM1&connectionFactory=

connectQueueManager(QM1)&initialContextFactory=com.ibm.mq.jms.Nojndi");
```

2. Create the service host with an instance of the required service class:

```
ServiceHost service = new ServiceHost(typeof(MyService));
```

3. Open the service:

```
service.AddServiceEndpoint(typeof(IMyServiceContract), binding, address);
service.Open();
...
service.Close();
```

Exposing metadata using an HTTP endpoint

Instructions for exposing the metadata of a service which is configured to use the IBM MQ custom channel for WCF.

About this task

If the services metadata must be exposed (so that tools such as `svcutil` can access it directly from the running service rather than from an offline WSDL file for example) it must be done by exposing the services metadata with an HTTP endpoint. The following steps can be used to add this additional endpoint.

1. Add the base address of where the metadata must be exposed to the ServiceHost, for example:

```
ServiceHost service = new ServiceHost(typeof(TestService),
                      new Uri("http://localhost:8000/MyService"));
```

2. Add the following code to the ServiceHost before the service is opened:

```
ServiceMetadataBehavior metadataBehavior = new ServiceMetadataBehavior();
  metadataBehavior.HttpGetEnabled = true;
  service.Description.Behaviors.Add(metadataBehavior);
  service.AddServiceEndpoint(typeof(IMetadataExchange),
     MetadataExchangeBindings.CreateMexHttpBinding(), "mex");
```

Results

The metadata is now available at the following address: `http://localhost:8000/MyService`

Building client applications for WCF

Overview of generating and building Microsoft Windows Communication Foundation (WCF) client applications.

A client application can be created for a WCF service; client applications are typically generated by using the Microsoft ServiceModel Metadata Utility Tool (Svcutil.exe) to create the required configuration and proxy files which can be used directly by the application.

Generating a WCF client proxy and application configuration files using the svcutil tool with metadata from a running service

Instructions for using the Microsoft svcutil.exe tool to generate a client for a service which is configured to use the IBM MQ custom channel for WCF.

Before you begin

There are three prerequisites for using the svcutil tool to create to required configuration and proxy files which can be used directly by the application:

- The WCF service must be running before the svcutil tool is started.
- The WCF service must expose its metadata using an HTTP port in addition to the IBM MQ custom channel endpoint references to generate a client directly from a running service.
- The custom channel must be registered in the configuration data for svcutil.

About this task

The following steps explain how to generate a client for a service which is configured to use the IBM MQ custom channel, but also exposes its metadata at run time via a separate HTTP port:

1. Start the WCF service (The service must be running before the svcutil tool is started).
2. Add the details from the `svcutil.exe` config file from the root of the installation, into the active svcutil configuration file, typically `C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin\svcutil.exe.config` so svcutil recognizes the IBM MQ custom channel.
3. Run svcutil from a command prompt, for example:

```
svcutil /language:C# /r: installlocation\bin\IBM.XMS.WCF.dll
/config:app.config http://localhost:8000/IBM.XMS.WCF/samples
```

4. Copy the generated `app.config` and `YourService.cs` files to the Microsoft Visual studio client project.

What to do next

If the services metadata cannot be directly retrieved, svcutil can be used to generate the client files from wsdl instead. For more information see: "Generating a WCF client proxy and application configuration files using the svcutil tool with WSDL" on page 1240

Generating a WCF client proxy and application configuration files using the svcutil tool with WSDL

Instructions for generating WCF clients from WSDL if the metadata of the service is unavailable.

If the metadata of the service cannot be directly retrieved to generate a client from the metadata from a running service, then, svcutil can be used to generate the client files from WSDL instead. The following modifications must be made to the WSDL to specify that the IBM MQ custom channel is to be used:

1. Add the following namespace definitions and policy information:

```
<wsdl:definitions
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">

        <wsp:Policy wsu:Id="CustomBinding_IWMQSampleContract_policy">
            <wsp:ExtactlyOne>
                <wsp:All>
                    <xms:xms xmlns:xms="http://sample.schemas.ibm.com/policy/xms" />
                </wsp:All>
            </wsp:ExactlyOne>
        </wsp:Policy>

...

</wsdl:definitions>
```

2. Modify the bindings section to refer to the new policy section and remove any `transport` definition from the underlying binding element:

```
<wsdl:definitions ...>

        <wsdl:binding ...>
                <wsp:PolicyReference URI="#CustomerBinding_IWMQSampleContract_policy" />
                <[soap]:binding ... transport="" />
                ...
        </wsdl:binding>
</wsdl:definitions>
```

3. Run svcutil from a command prompt, for example:

```
svcutil /language:C# /r:  MQ_INSTALLATION_PATH\bin\IBM.XMS.WCF.dll
/config:app.config  MQ_INSTALLATION_PATH\src\samples\WMQAxis\default\service
\soap.server.stockQuoteAxis_Wmq.wsdl
```

Building WCF client applications using a client proxy with an application configuration file

Before you begin

Create or edit an application configuration file for the client, as described in: "Creating a WCF custom channel administratively by supplying binding and endpoint information in an application configuration file" on page 1230

About this task

Instantiate and open an instance of the client proxy. The parameter passed to the generated proxy must be the same as the endpoint name specified in the client configuration file, for example `Endpoint_WMQ`:

```
MyClientProxy myClient = new MyClientProxy("Endpoint_WMQ");
```

```
            try {
                   myClient.myMethod("HelloWorld!");
                   myClient.Close();
            }
            catch (TimeoutException e) {
                Console.Out.WriteLine(e);
                myClient.Abort();
            }
            catch (CommunicationException e) {
                Console.Out.WriteLine(e);
                myClient.Abort();
            }
            catch (Exception e) {
                Console.Out.WriteLine(e);
                myClient.Abort();
            }
```

Building WCF client applications using a client proxy with programmatic configuration

Before you begin

1. Add a reference to the custom channel `IBM.XMS.WCF.dll` file to the project. The `IBM.XMS.WCF.dll` is in the *`WMQInstallDir`*`\bin` directory where *WMQInstallDir* is the directory that IBM MQ is installed in.
2. Add a *using* statement to the `IBM.XMS.WCF` namespace, for example: `using IBM.XMS.WCF`
3. Create an instance of th' binding element and endpoint of the channel as described in: "Creating a WCF custom channel by suppling binding and endpoint information programmatically" on page 1232

About this task

If changes to the binding properties of the channel are required, complete the following steps.

1. Add the binding properties to `transportBindingElement` as shown in the following figure:

```
SoapJmsIbmTransportBindingElement transportBindingElement = new
SoapJmsIbmTransportBindingElement();
Binding binding = new CustomBinding(new TextMessageEncodingBindingElement(),
transportBindingElement);
EndpointAddress address =
  new EndpointAddress("jms:/queue?destination=SampleQ@QM1&connectionFactory=
     connectQueueManager(QM1)&initialContextFactory=com.ibm.mq.jms.Nojndi");
```

2. Create the client proxy as shown in the following figure, where *binding* and *endpoint address* are the binding and endpoint address configured in step 1 and passed in:

```
 MyClientProxy myClient = new MyClientProxy(binding, endpoint address);
            try {
                   myClient.myMethod("HelloWorld!");
                   myClient.Close();
            }
            catch (TimeoutException e) {
                Console.Out.WriteLine(e);
                myClient.Abort();
            }
            catch (CommunicationException e) {
                Console.Out.WriteLine(e);
                                  myClient.Abort();
            }
            catch (Exception e) {
                Console.Out.WriteLine(e);
                myClient.Abort();
            }
```

Using the WCF samples

The Windows Communication Foundation (WCF) samples provide some simple examples of how the IBM MQ custom channel can be used.

To build the sample projects, either the Microsoft.NET 3.5 SDK, or Microsoft Visual Studio 2008 is needed.

Simple one-way client and server WCF sample

This sample demonstrates the IBM MQ custom channel being used to start a Windows Communication foundation (WCF) service from a WCF client using a one-way channel shape.

About this task

The service implements a single method which outputs a string to the console. The client has been generated by using the `svcutil` tool to retrieve the service metadata from a separately exposed HTTP endpoint as described in "Generating a WCF client proxy and application configuration files using the svcutil tool with metadata from a running service" on page 1239

The sample has been configured with specific resource names as described in the following procedure. If you must change the resource names, then you must also change the corresponding value on the client application in the *MQ_INSTALLATION_PATH* `\tools\dotnet\samples\cs\wcf\samples\WCF\oneway\client\app.config` file, and on the service application in the *MQ_INSTALLATION_PATH* `\tools\dotnet\samples\cs\wcf\samples\WCF\oneway\service\TestServices.cs` file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ. For more information about formatting the JMS endpoint URI, see *IBM MQ Transport for SOAP* in the IBM MQ product documentation. If you need to modify the sample solution and source, then you need an IDE, for example, Microsoft Visual Studio 8 or higher.

Procedure

1. Create a queue manager called *QM1*
2. Create a queue destination called *SampleQ*
3. Start the service so the listener is waiting for messages: Run the *MQ_INSTALLATION_PATH* `\tools\dotnet\samples\cs\wcf\samples\WCF\oneway\service\bin\Release\TestService.exe` file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ.
4. Run the client once: Run the *MQ_INSTALLATION_PATH* `\tools\dotnet\samples\cs\wcf\samples\WCF\oneway\client\bin\Release\TestClient.exe` file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ.

   The client application loops five times sending five messages to *SampleQ*

Results

The service application gets the messages from *SampleQ* and displays `Hello World` on the screen five times.

What to do next

Simple request-reply client and server WCF sample

This sample demonstrates the IBM MQ custom channel being used to start a Windows Communication foundation (WCF) service from a WCF client using a request-reply channel shape.

About this task

This service provides some simple calculator methods to add and subtract two numbers, and then return the result. The client has been generated by using the `svcutil` tool to retrieve the service metadata from a separately exposed HTTP endpoint as described in "Generating a WCF client proxy and application configuration files using the svcutil tool with metadata from a running service" on page 1239

The sample has been configured with specific resource names as in the following procedure described. If you need to change the resource names, then you also need to change the corresponding value on the client application in the *MQ_INSTALLATION_PATH* `\Tools\wcf\samples\WCF\requestreply\client\app.config` file, and on the service application in the *MQ_INSTALLATION_PATH* `\Tools\wcf\samples\WCF\requestreply\service\RequestReplyService.cs` file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ. For more information about formatting the JMS endpoint URI, see *IBM MQ Transport for SOAP* in the IBM MQ product documentation. If you need to modify the sample solution and source, then you need an IDE, for example, Microsoft Visual Studio 8 or higher.

Procedure

1. Create a queue manager called *QM1*
2. Create a queue destination called *SampleQ*
3. Create a queue destination called *SampleReplyQ*
4. Start the service so the listener is waiting for messages: Run the *MQ_INSTALLATION_PATH* `\Tools\wcf\samples\WCF\requestreply\service\bin\Release\SimpleRequestReply_Service.exe` file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ.
5. Run the client once: Run the *MQ_INSTALLATION_PATH* `\Tools\wcf\samples\WCF\requestreply\client\bin\Release\SimpleRequestReply_Client.exe` file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ.

Results

When the client has been run, the following process is started and repeats four times so a total of five messages are sent each way:

1. The client puts a request message on *SampleQ* and waits for a response.
2. The service gets the request message from *SampleQ*.
3. The service adds and subtracts some values using the contents of the message.
4. The service then puts the results into a message on *SampleReplyQ*, and waits for the client to put a new message.
5. The client gets the message from *SampleReplyQ* and displays the results on the screen.

What to do next

WCF client to a .NET service hosted by IBM MQ sample

Sample client applications and sample service proxy applications are supplied for both .NET and Java. The samples are based on a Stock Quote service that takes a request for a stock quote and then provides the stock quote.

Before you begin

The sample requires that the .NET SOAP over JMS service hosting environment is correctly installed and configured in IBM MQ and is accessible from a local queue manager.

When the .NET SOAP over JMS service hosting environment is correctly installed and configured in IBM MQ and is accessible from a local queue manager, additional configuration steps must be completed.

1. Set the **WMQSOAP_HOME** environment variable to the IBM MQ installation directory, for example: `C:\Program Files\IBM\MQ`
2. Ensure that the Java compiler `javac` is available and on the PATH.
3. Copy the file `axis.jar` from the `prereqs/axis` directory of the installation image to the IBM MQ production directory, for example: `C:\Program Files\IBM\MQ\java\lib\soap`

4. Add to the PATH: *MQ_INSTALLATION_PATH*\Java\lib where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed, for example: C:\Program Files\IBM\MQ
5. Ensure that the location of .NET is specified correctly in *MQ_INSTALLATION_PATH*\bin\amqwcallWSDL.cmd where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed, for example: C:\Program Files\IBM\MQ. The location of .NET can be specified for example: set msfwdir=%ProgramFiles%\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin

When the previous steps are complete, test and run the service:

1. Navigate to your SOAP over JMS working directory.
2. Enter one of the following commands to run the verification test and leave the service listener running:
   - For .NET: *MQ_INSTALLATION_PATH*\Tools\soap\samples\runivt dotnet hold where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed.
   - For AXIS: *MQ_INSTALLATION_PATH*\Tools\soap\samples\runivt Dotnet2AxisClient hold where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed.

The hold argument keeps the listeners running after the test completes.

If errors are reported during this configuration, you can remove all the changes so that the procedure may be restarted in the following way:

1. Delete the generated SOAP over JMS directory.
2. Delete the queue manager.

About this task

This sample demonstrates a connection from a WCF client to the .NET SOAP over JMS sample service provided in IBM MQ using a one-way channel shape. The service implements a simple StockQuote example, which outputs a text string to the console.

The client has been generated by using WSDL to generate client files as described in "Generating a WCF client proxy and application configuration files using the svcutil tool with WSDL" on page 1240

The sample has been configured with specific resource names as described in the following procedure. If you need to change the resource names, then you must also change the corresponding value on the client application in the *MQ_INSTALLATION_PATH* \tools\wcf\samples\WMQNET\default\client\app.config file, and on the service application in the *MQ_INSTALLATION_PATH* \tools\wcf\samples\WMQNET\default\service\WmqDefaultSample_StockQuoteDotNet.wsdl file, where *MQ_INSTALLATION_PATH* represents the installation directory for IBM MQ. For more information about formatting the JMS endpoint URI, see *IBM MQ Transport for SOAP* in the IBM MQ product documentation.

Procedure

Run the client once: Run the *MQ_INSTALLATION_PATH* \tools\wcf\samples\WMQNET\default\client\bin\Release\TestClient.exe file, where *MQ_INSTALLATION_PATH* represents the installation directory for IBM MQ.

The client application loops five times sending five messages to the sample queue.

Results

The service application gets the messages from the sample queue and displays Hello World five times on the screen.

WCF client to an Axis Java service hosted by IBM MQ sample

Sample client applications and sample service proxy applications are supplied for both Java and .NET. The samples are based on a Stock Quote service that takes a request for a stock quote and then provides the stock quote.

Before you begin

This sample requires that the .NET SOAP over JMS service hosting environment is correctly installed and configured in IBM MQ and is accessible from a local queue manager.

When the .NET SOAP over JMS service hosting environment is correctly installed and configured in IBM MQ and is accessible from a local queue manager, additional configuration steps must be completed.

1. Set the **WMQSOAP_HOME** environment variable to the IBM MQ installation directory, for example: `C:\Program Files\IBM\MQ`
2. Ensure that the Java compiler `javac` is available and on the PATH.
3. Copy the file `axis.jar` from the `prereqs/axis` directory of the installation image to the IBM MQ installation directory.
4. Add to the PATH: *MQ_INSTALLATION_PATH*`\Java\lib` where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed, for example: `C:\Program Files\IBM\MQ`
5. Ensure that the location of .NET is specified correctly in *MQ_INSTALLATION_PATH*`\bin\amqwcallWSDL.cmd` where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed, for example: `C:\Program Files\IBM\MQ`. The location of .NET can be specified for example: `set msfwdir=%ProgramFiles%\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin`

When the previous steps are complete, test and run the service:

1. Navigate to your SOAP over JMS working directory.
2. Enter one of the following commands to run the verification test and leave the service listener running:
   - For .NET: *MQ_INSTALLATION_PATH*`\Tools\soap\samples\runivt dotnet hold` where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed.
   - For AXIS: *MQ_INSTALLATION_PATH*`\Tools\soap\samples\runivt Dotnet2AxisClient hold` where *MQ_INSTALLATION_PATH* represents the directory where IBM MQ is installed.

The `hold` argument keeps the listeners running after the test completes.

If errors are reported during this configuration, you can remove all the changes so that the procedure is restarted in the following way:

1. Delete the generated SOAP over JMS directory.
2. Delete the queue manager.

About this task

The sample demonstrates a connection from a WCF client to the Axis Java SOAP over JMS sample service provided in IBM MQ using a one-way channel shape. The service implements a simple StockQuote example, which outputs a text string to a file which is saved in the current directory.

The client has been generated by using WSDL to generate client files as described in "Generating a WCF client proxy and application configuration files using the svcutil tool with WSDL" on page 1240

The sample has been configured with specific resource names as described in this paragraph. If you need to change the resource names, then you must also change the corresponding value on the client application in the *MQ_INSTALLATION_PATH* `\tools\wcf\samples\WMQAxis\default\client\app.config` file, and on the service application in the *MQ_INSTALLATION_PATH* `\tools\wcf\samples\WMQAxis\default\service\WmqDefaultSample_StockQuoteDotNet.wsdl` file, where *MQ_INSTALLATION_PATH* represents the installation directory for IBM MQ.

Procedure

Run the client once: Run the *MQ_INSTALLATION_PATH* `\tools\wcf\samples\WMQAxis\default\client\bin\Release\TestClient.exe` file, where *MQ_INSTALLATION_PATH* represents the installation directory for IBM MQ.

The client application loops five times sending five messages to the sample queue.

Results

The service application gets the messages from the sample queue and adds `Hello World` five times to a file in the current directory.

WCF client to Java service hosted by WebSphere Application Server sample

Sample client applications and sample service proxy applications are supplied for WebSphere Application Server 6. A request-response service is also provided.

Before you begin

This sample requires that the following IBM MQ configuration is used:

Table 195. IBM MQ required configuration

Object	**Required name**
Queue manager	QM1
Local queue	HelloWorld
Local queue	HelloWorldReply

This sample also requires that a WebSphere Application Server 6 hosting environment is correctly installed and configured. WebSphere Application Server 6 uses a bindings mode connection to connect to IBM MQ by default. Therefore WebSphere Application Server 6 must be installed on the same machine as the queue manager.

After the WAS environment is configured, the following additional configuration steps must be completed:

1. Create the following JNDI objects in the WebSphere Application Server JNDI repository:
   a. A JMS queue destination called `HelloWorld`
      - Set the JNDI name to `jms/HelloWorld`
      - Set the queue name to `HelloWorld`
   b. A JMS queue connection factory called `HelloWorldQCF`
      - Set the JNDI name to `jms/HelloWorldQCF`
      - Set the queue manager name to `QM1`
   c. A JMS queue connection factory called `WebServicesReplyQCF`
      - Set the JNDI name to `jms/WebServicesReplyQCF`
      - Set the queue manager name to `QM1`
2. Create a Message Listener Port called `HelloWorldPort` in WebSphere Application Server with the following configuration:
   - Set the connection factory JNDI name to `jms/HelloWorldQCF`
   - Set the destination JNDI name to `jms/HelloWorld`
3. Install the web service `HelloWorldEJBEAR.ear` application to your WebSphere Application Server as follows:
   a. Click **Applications** > **New Application** > **New Enterprise Application**.

b. Navigate to *MQ_INSTALLATION_PATH*\tools\wcf\samples\WAS\HelloWorldsEJBEAR.ear where *MQ_INSTALLATION_PATH* is the installation directory of IBM MQ.

c. Do not change any of the default option in the wizard, and restart the application server after the application has been installed.

When the WAS configuration is complete, test the service by running it once:

1. Navigate to your Soap over JMS working directory.
2. Enter this command to run the sample: *MQ_INSTALLATION_PATH* \tools\wcf\samples\WAS\TestClient.exe where *MQ_INSTALLATION_PATH* is the installation directory of IBM MQ.

About this task

The sample demonstrates a connection from a WCF client to the WebSphere Application Server SOAP over JMS sample service provided in the WCF samples included in IBM MQ, using a request-response channel shape. Messages flow between WCF and the WebSphere Application Server using IBM MQ queues. The service implements the HelloWorld(...) method, which takes a string and returns a greeting to the client.

The client has been generated by using the svcutil tool to retrieve the service metadata from a separately exposed HTTP endpoint as described in "Generating a WCF client proxy and application configuration files using the svcutil tool with metadata from a running service" on page 1239

The sample has been configured with specific resource names as described in the following procedure. If you need to change the resource names, then you must also change the corresponding value on the client application in the *MQ_INSTALLATION_PATH* \tools\wcf\samples\WAS\default\client\app.config file, and on the service application in the *MQ_INSTALLATION_PATH* \tools\wcf\samples\WAS\HelloWorldsEJBEAR.ear where *MQ_INSTALLATION_PATH* is the installation directory of IBM MQ.

The service and client are based upon the service and client outlined in the IBM Developer article *Building a JMS web service using SOAP over JMS and WebSphere Studio*. For more information about developing SOAP over JMS web services that are compatible with the IBM MQ WCF custom channel, see https://www.ibm.com/developerworks/websphere/library/techarticles/0402_du/0402_du.html.

Procedure

Run the client once: Run the *MQ_INSTALLATION_PATH* \tools\wcf\samples\WAS\default\client\bin\Release\TestClient.exe file, where *MQ_INSTALLATION_PATH* is the installation directory for IBM MQ.

The client application starts both of the service methods at the same time, sending two messages to the sample queue.

Results

The service application gets the messages from the sample queue and provides a response to the HelloWorld(...) method call which the client application outputs to the console.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

This book contains information on intended programming interfaces that allow the customer to write programs to obtain the services of IBM MQ.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Important: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks

IBM, the IBM logo, ibm.com®, are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at "Copyright and trademark information"www.ibm.com/legal/copytrade.shtml. Other product and service names might be trademarks of IBM or other companies.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

This product includes software developed by the Eclipse Project (https://www.eclipse.org/).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Part Number:

(1P) P/N: