7.5

Administering IBM WebSphere MQ

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 159.

This edition applies to version 7 release 5 of IBM® WebSphere® MQ and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Administering IBM WebSphere MQ

Administering queue managers and associated resources includes the tasks that you perform frequently to activate and manage those resources. Choose the method you prefer to administer your queue managers and associated resources.

You can administer IBM WebSphere MQ objects locally or remotely, see "Local and remote administration" on page 7.

There are a number of different methods that you can use to create and administer your queue managers and their related resources in IBM WebSphere MQ. These methods include command-line interfaces, a graphical user interface, and an administration API. See the sections and links in this topic for more information about each of these interfaces.

There are different sets of commands that you can use to administer IBM WebSphere MQ depending on your platform:

- "IBM WebSphere MQ control commands" on page 5
- "IBM WebSphere MQ Script (MQSC) commands" on page 5
- "Programmable Command Formats (PCFs)" on page 6

There are also the other following options for creating and managing IBM WebSphere MQ objects:

- "The IBM WebSphere MQ Explorer" on page 6
- "The Windows Default Configuration application" on page 6
- "The Microsoft Cluster Service (MSCS)" on page 7

You can automate some administration and monitoring tasks for both local and remote queue managers by using PCF commands. These commands can also be simplified through the use of the IBM WebSphere MQ Administration Interface (MQAI) on some platforms. For more information about automating administration tasks, see "Automating administration tasks" on page 8.

IBM WebSphere MQ control commands

Control commands allow you to perform administrative tasks on queue managers themselves.

IBM WebSphere MQ for Windows, UNIX and Linux® systems provides the *control commands* that you issue at the system command line.

The control commands are described in Creating and managing queue managers. For the command reference for the control commands, see IBM WebSphere MQ Control commands.

IBM WebSphere MQ Script (MQSC) commands

Use MQSC commands to manage queue manager objects, including the queue manager itself, queues, process definitions, namelists, channels, client connection channels, listeners, services, and authentication information objects.

You issue MQSC commands to a queue manager by using the `runmqsc` command. You can do this interactively, issuing commands from a keyboard, or you can redirect the standard input device (stdin) to run a sequence of commands from an ASCII text file. In both cases, the format of the commands is the same.

You can run the `runmqsc` command in three modes, depending on the flags set on the command:

- *Verification mode*, where the MQSC commands are verified on a local queue manager, but are not run
- *Direct mode*, where the MQSC commands are run on a local queue manager
- *Indirect mode*, where the MQSC commands are run on a remote queue manager

Object attributes specified in MQSC commands are shown in this section in uppercase (for example, RQMNAME), although they are not case-sensitive. MQSC command attribute names are limited to eight characters.

MQSC commands are available on all platforms. MQSC commands are summarized in Comparing command sets.

On Windows, UNIX or Linux, you can use the MQSC as single commands issued at the system command line. To issue more complicated, or multiple commands, the MQSC can be built into a file that you run from the Windows, UNIX or Linux system command line. MQSC can be sent to a remote queue manager. For full details, see MQSC reference.

"Script (MQSC) Commands" on page 72 contains a description of each MQSC command and its syntax.

See "Performing local administration tasks using MQSC commands" on page 71 for more information about using MQSC commands in local administration.

Programmable Command Formats (PCFs)

Programmable Command Formats (PCFs) define command and reply messages that can be exchanged between a program and any queue manager (that supports PCFs) in a network. You can use PCF commands in a systems management application program for administration of IBM WebSphere MQ objects: authentication information objects, channels, channel listeners, namelists, process definitions, queue managers, queues, services, and storage classes. The application can operate from a single point in the network to communicate command and reply information with any queue manager, local, or remote, using the local queue manager.

For more information about PCFs, see "Introduction to Programmable Command Formats" on page 9.

For definition of PCFs and structures for the commands and responses, see Programmable command formats reference.

The IBM WebSphere MQ Explorer

Using the IBM WebSphere MQ Explorer, you can perform the following actions:

- Define and control various resources including queue managers, queues, process definitions, namelists, channels, client connection channels, listeners, services, and clusters.
- Start or stop a local queue manager and its associated processes.
- View queue managers and their associated objects on your workstation or from other workstations.
- Check the status of queue managers, clusters, and channels.
- Check to see which applications, users, or channels have a particular queue open, from the queue status.

On Windows and Linux systems, you can start IBM WebSphere MQ Explorer by using the system menu, the `MQExplorer` executable file, or the **`strmqcfg`** command.

On Linux, to start the IBM WebSphere MQ Explorer successfully, you must be able to write a file to your home directory, and the home directory must exist.

You can use IBM WebSphere MQ Explorer to administer remote queue managers on other platforms including z/OS®, for details and to download the SupportPac MS0T, see https://www.ibm.com/support/docview.wss?uid=swg24021041.

See "Administration using the IBM WebSphere MQ Explorer" on page 56 for more information.

The Windows Default Configuration application

You can use the Windows Default Configuration program to create a *starter* (or default) set of IBM WebSphere MQ objects. A summary of the default objects created is listed in Table 1: Objects created by the Windows default configuration application.

The Microsoft Cluster Service (MSCS)

Microsoft Cluster Service (MSCS) enables you to connect servers into a *cluster*, giving higher availability of data and applications, and making it easier to manage the system. MSCS can automatically detect and recover from server or application failures.

It is important not to confuse clusters in the MSCS sense with IBM WebSphere MQ clusters. The distinction is:

IBM WebSphere MQ clusters
are groups of two or more queue managers on one or more computers, providing automatic interconnection, and allowing queues to be shared among them for load balancing and redundancy.

MSCS clusters
Groups of computers, connected together and configured in such a way that, if one fails, MSCS performs a *failover*, transferring the state data of applications from the failing computer to another computer in the cluster and reinitiating their operation there.

Supporting the Microsoft Cluster Service (MSCS) provides detailed information about how to configure your IBM WebSphere MQ for Windows system to use MSCS.

Related concepts

WebSphere MQ technical overview

"Administering local IBM WebSphere MQ objects" on page 67
This section tells you how to administer local IBM WebSphere MQ objects to support application programs that use the Message Queue Interface (MQI). In this context, local administration means creating, displaying, changing, copying, and deleting IBM WebSphere MQ objects.

"Administering remote IBM WebSphere MQ objects" on page 103

Considerations when contact is lost with the XA resource manager

Related tasks

Planning

Configuring

Related reference

Transactional support scenarios

Local and remote administration

You can administer WebSphere MQ objects locally or remotely.

Local administration means carrying out administration tasks on any queue managers you have defined on your local system. You can access other systems, for example through the TCP/IP terminal emulation program **telnet**, and carry out administration there. In WebSphere MQ, you can consider this as local administration because no channels are involved, that is, the communication is managed by the operating system.

WebSphere MQ supports administration from a single point of contact through what is known as *remote administration*. This allows you to issue commands from your local system that are processed on another system and applies also to the WebSphere MQ Explorer. For example, you can issue a remote command to change a queue definition on a remote queue manager. You do not have to log on to that system, although you do need to have the appropriate channels defined. The queue manager and command server on the target system must be running.

Some commands cannot be issued in this way, in particular, creating or starting queue managers and starting command servers. To perform this type of task, you must either log onto the remote system and issue the commands from there or create a process that can issue the commands for you. This restriction applies also to the WebSphere MQ Explorer.

"Administering remote IBM WebSphere MQ objects" on page 103 describes the subject of remote administration in greater detail.

How to use IBM WebSphere MQ control commands

This section describes how to use the IBM WebSphere MQ control commands.

If you want to issue control commands, your user ID must be a member of the mqm group. For more information about this, see Authority to administer WebSphere MQ on UNIX, Linux and Windows systems . In addition, note the following environment-specific information:

WebSphere MQ for Windows
All control commands can be issued from a command line. Command names and their flags are not case sensitive: you can enter them in uppercase, lowercase, or a combination of uppercase and lowercase. However, arguments to control commands (such as queue names) are case sensitive.

In the syntax descriptions, the hyphen (-) is used as a flag indicator. You can use the forward slash (/) instead of the hyphen.

WebSphere MQ for UNIX and Linux systems
All WebSphere MQ control commands can be issued from a shell. All commands are case-sensitive.

A subset of the control commands can be issued using the IBM WebSphere MQ Explorer.

For more information, see The WebSphere MQ control commands

Automating administration tasks

You might decide that it would be beneficial to your installation to automate some administration and monitoring tasks. You can automate administration tasks for both local and remote queue managers using programmable command format (PCF) commands. This section assumes that you have experience of administering WebSphere MQ objects.

PCF commands

WebSphere MQ programmable command format (PCF) commands can be used to program administration tasks into an administration program. In this way, from a program you can manipulate queue manager objects (queues, process definitions, namelists, channels, client connection channels, listeners, services, and authentication information objects), and even manipulate the queue managers themselves.

PCF commands cover the same range of functions provided by MQSC commands. You can write a program to issue PCF commands to any queue manager in the network from a single node. In this way, you can both centralize and automate administration tasks.

Each PCF command is a data structure that is embedded in the application data part of a WebSphere MQ message. Each command is sent to the target queue manager using the MQI function `MQPUT` in the same way as any other message. Providing the command server is running on the queue manager receiving the message, the command server interprets it as a command message and runs the command. To get the replies, the application issues an `MQGET` call and the reply data is returned in another data structure. The application can then process the reply and act accordingly.

Note: Unlike MQSC commands, PCF commands and their replies are not in a text format that you can read.

Briefly, these are some of the things needed to create a PCF command message:

Message descriptor
This is a standard WebSphere MQ message descriptor, in which:

- Message type (*MsgType*) is MQMT_REQUEST.
- Message format (*Format*) is MQFMT_ADMIN.

Application data
Contains the PCF message including the PCF header, in which:

- The PCF message type (*Type*) specifies MQCFT_COMMAND.

- The command identifier specifies the command, for example, *Change Queue* (MQCMD_CHANGE_Q).

For a complete description of the PCF data structures and how to implement them, see "Introduction to Programmable Command Formats" on page 9.

PCF object attributes

Object attributes in PCF are not limited to eight characters as they are for MQSC commands. They are shown in this guide in italics. For example, the PCF equivalent of RQMNAME is *RemoteQMgrName*.

Escape PCFs

Escape PCFs are PCF commands that contain MQSC commands within the message text. You can use PCFs to send commands to a remote queue manager. For more information about escape PCFs, see Escape.

Introduction to Programmable Command Formats

Programmable Command Formats (PCFs) define command and reply messages that can be exchanged between a program and any queue manager (that supports PCFs) in a network. PCFs simplify queue manager administration and other network administration. They can be used to solve the problem of complex administration of distributed networks especially as networks grow in size and complexity.

The Programmable Command Formats described in this product documentation are supported by:

- IBM WebSphere MQ for AIX®
- IBM WebSphere MQ for HP-UX
- IBM WebSphere MQ for Linux
- IBM WebSphere MQ for Solaris
- IBM WebSphere MQ for Windows
- IBM WebSphere MQ for HP Integrity NonStop Server

The problem PCF commands solve

The administration of distributed networks can become complex. The problems of administration continue to grow as networks increase in size and complexity.

Examples of administration specific to messaging and queuing include:

- Resource management.

  For example, queue creation and deletion.
- Performance monitoring.

  For example, maximum queue depth or message rate.
- Control.

  For example, tuning queue parameters such as maximum queue depth, maximum message length, and enabling and disabling queues.
- Message routing.

  Definition of alternative routes through a network.

WebSphere MQ PCF commands can be used to simplify queue manager administration and other network administration. PCF commands allow you to use a single application to perform network administration from a single queue manager within the network.

What are PCFs?

PCFs define command and reply messages that can be exchanged between a program and any queue manager (that supports PCFs) in a network. You can use PCF commands in a systems management application program for administration of WebSphere MQ objects: authentication information objects, channels, channel listeners, namelists, process definitions, queue managers, queues, services, and storage classes. The application can operate from a single point in the network to communicate command and reply information with any queue manager, local, or remote, using the local queue manager.

Each queue manager has an administration queue with a standard queue name and your application can send PCF command messages to that queue. Each queue manager also has a command server to service the command messages from the administration queue. PCF command messages can therefore be processed by any queue manager in the network and the reply data can be returned to your application, using your specified reply queue. PCF commands and reply messages are sent and received using the normal Message Queue Interface (MQI).

For a list of the available PCF commands, including their parameters, see Definitions of the Programmable Command Formats.

Using Programmable Command Formats

You can use PCFs in a systems management program for WebSphere MQ remote administration.

This section includes:

- “PCF command messages” on page 10
- “Responses” on page 12
- Rules for naming IBM WebSphere MQ objects
- “Authority checking for PCF commands” on page 15

PCF command messages

PCF command messages consist of a PCF header, parameters identified in that header and also user-defined message data. The messages are issued using Message Queue interface calls.

Each command and its parameters are sent as a separate command message containing a PCF header followed by a number of parameter structures; for details of the PCF header, see MQCFH - PCF header, and for an example of a parameter structure, see MQCFST - PCF string parameter. The PCF header identifies the command and the number of parameter structures that follow in the same message. Each parameter structure provides a parameter to the command.

Replies to the commands, generated by the command server, have a similar structure. There is a PCF header, followed by a number of parameter structures. Replies can consist of more than one message but commands always consist of one message only.

On platforms other than z/OS, the queue to which the PCF commands are sent is always called the SYSTEM.ADMIN.COMMAND.QUEUE.

How to issue PCF command messages

Use the normal Message Queue Interface (MQI) calls, MQPUT, MQGET, and so on, to put and retrieve PCF command and response messages to and from their queues.

Note:

Ensure that the command server is running on the target queue manager for the PCF command to process on that queue manager.

For a list of supplied header files, see WebSphere MQ COPY, header, include and module files.

Message descriptor for a PCF command

The WebSphere MQ message descriptor is fully documented in MQMD - Message descriptor.

A PCF command message contains the following fields in the message descriptor:

Report
Any valid value, as required.

MsgType
This field must be MQMT_REQUEST to indicate a message requiring a response.

Expiry
Any valid value, as required.

Feedback
Set to MQFB_NONE

Encoding
If you are sending to Windows, UNIX or Linux systems, set this field to the encoding used for the message data; conversion is performed if necessary.

CodedCharSetId
If you are sending to Windows, UNIX or Linux systems, set this field to the coded character-set identifier used for the message data; conversion is performed if necessary.

Format
Set to MQFMT_ADMIN.

Priority
Any valid value, as required.

Persistence
Any valid value, as required.

MsgId
The sending application can specify any value, or MQMI_NONE can be specified to request the queue manager to generate a unique message identifier.

CorrelId
The sending application can specify any value, or MQCI_NONE can be specified to indicate no correlation identifier.

ReplyToQ
The name of the queue to receive the response.

ReplyToQMgr
The name of the queue manager for the response (or blank).

Message context fields
These fields can be set to any valid values, as required. Normally the Put message option MQPMO_DEFAULT_CONTEXT is used to set the message context fields to the default values.

If you are using a version-2 MQMD structure, you must set the following additional fields:

GroupId
Set to MQGI_NONE

MsgSeqNumber
Set to 1

Offset
Set to 0

MsgFlags
Set to MQMF_NONE

OriginalLength
Set to MQOL_UNDEFINED

Sending user data

The PCF structures can also be used to send user-defined message data. In this case the message descriptor *Format* field must be set to MQFMT_PCF.

Sending and receiving PCF messages in a specified queue

Sending PCF messages to a specified queue

To send a message to a specified queue, the mqPutBag call converts the contents of the specified bag into a PCF message and sends the message to the specified queue. The contents of the bag are left unchanged after the call.

As input to this call, you must supply:

- An MQI connection handle.
- An object handle for the queue on which the message is to be placed.
- A message descriptor. For more information about the message descriptor, see MQMD - Message descriptor.
- Put Message Options using the MQPMO structure. For more information about the MQPMO structure, see MQPMO - Put-message options.
- The handle of the bag to be converted to a message.

  Note: If the bag contains an administration message and the mqAddInquiry call was used to insert values into the bag, the value of the MQIASY_COMMAND data item must be an INQUIRE command recognized by the MQAI.

For a full description of the mqPutBag call, see mqPutBag.

Receiving PCF messages from a specified queue

To receive a message from a specified queue, the mqGetBag call gets a PCF message from a specified queue and converts the message data into a data bag.

As input to this call, you must supply:

- An MQI connection handle.
- An object handle of the queue from which the message is to be read.
- A message descriptor. Within the MQMD structure, the `Format` parameter must be MQFMT_ADMIN, MQFMT_EVENT, or MQFMT_PCF.

  Note: If the message is received within a unit of work (that is, with the MQGMO_SYNCPOINT option) and the message has an unsupported format, the unit of work can be backed out. The message is then reinstated on the queue and can be retrieved using the MQGET call instead of the mqGetBag call. For more information about the message descriptor, see MQGMO - Get-message options.
- Get Message Options using the MQGMO structure. For more information about the MQGMO structure, see MQMD - Message Descriptor.
- The handle of the bag to contain the converted message.

For a full description of the mqGetBag call, see mqGetBag.

Responses

In response to each command, the command server generates one or more response messages. A response message has a similar format to a command message.

The PCF header has the same command identifier value as the command to which it is a response (see MQCFH - PCF header for details). The message identifier and correlation identifier are set according to the report options of the request.

If the PCF header type of the command message is MQCFT_COMMAND, standard responses only are generated. Such commands are supported on all platforms except z/OS. Older applications do not support PCF on z/OS; the WebSphere MQ Windows Explorer is one such application (however, the Version 6.0 or later IBM WebSphere MQ Explorer does support PCF on z/OS).

If the PCF header type of the command message is MQCFT_COMMAND_XR, either extended or standard responses are generated. Such commands are supported on z/OS and some other platforms. Commands issued on z/OS generate only extended responses. On other platforms, either type of response might be generated.

If a single command specifies a generic object name, a separate response is returned in its own message for each matching object. For response generation, a single command with a generic name is treated as multiple individual commands (except for the control field MQCFC_LAST or MQCFC_NOT_LAST). Otherwise, one command message generates one response message.

Certain PCF responses might return a structure even when it is not requested. This structure is shown in the definition of the response (Definitions of the Programmable Command Formats) as *always returned*. The reason that, for these responses, it is necessary to name the objects in the response to identify which object the data applies.

Message descriptor for a response

A response message has the following fields in the message descriptor:

MsgType
This field is MQMT_REPLY.

MsgId
This field is generated by the queue manager.

CorrelId
This field is generated according to the report options of the command message.

Format
This field is MQFMT_ADMIN.

Encoding
Set to MQENC_NATIVE.

CodedCharSetId
Set to MQCCSI_Q_MGR.

Persistence
The same as in the command message.

Priority
The same as in the command message.

The response is generated with MQPMO_PASS_IDENTITY_CONTEXT.

Standard responses

Command messages with a header type of MQCFT_COMMAND, standard responses are generated. Such commands are supported on all platforms except z/OS.

There are three types of standard response:

- OK response
- Error response
- Data response

OK response

This response consists of a message starting with a command format header, with a *CompCode* field of MQCC_OK or MQCC_WARNING.

For MQCC_OK, the *Reason* is MQRC_NONE.

For MQCC_WARNING, the *Reason* identifies the nature of the warning. In this case the command format header might be followed by one or more warning parameter structures appropriate to this reason code.

In either case, for an inquire command further parameter structures might follow as described in the following sections.

Error response

If the command has an error, one or more error response messages are sent (more than one might be sent even for a command that would normally have only a single response message). These error response messages have MQCFC_LAST or MQCFC_NOT_LAST set as appropriate.

Each such message starts with a response format header, with a *CompCode* value of MQCC_FAILED and a *Reason* field that identifies the particular error. In general, each message describes a different error. In addition, each message has either zero or one (never more than one) error parameter structures following the header. This parameter structure, if there is one, is an MQCFIN structure, with a *Parameter* field containing one of the following:

- MQIACF_PARAMETER_ID

  The *Value* field in the structure is the parameter identifier of the parameter that was in error (for example, MQCA_Q_NAME).
- MQIACF_ERROR_ID

  This value is used with a *Reason* value (in the command format header) of MQRC_UNEXPECTED_ERROR. The *Value* field in the MQCFIN structure is the unexpected reason code received by the command server.
- MQIACF_SELECTOR

  This value occurs if a list structure (MQCFIL) sent with the command contains a duplicate selector or one that is not valid. The *Reason* field in the command format header identifies the error, and the *Value* field in the MQCFIN structure is the parameter value in the MQCFIL structure of the command that was in error.
- MQIACF_ERROR_OFFSET

  This value occurs when there is a data compare error on the Ping Channel command. The *Value* field in the structure is the offset of the Ping Channel compare error.
- MQIA_CODED_CHAR_SET_ID

  This value occurs when the coded character-set identifier in the message descriptor of the incoming PCF command message does not match that of the target queue manager. The *Value* field in the structure is the coded character-set identifier of the queue manager.

The last (or only) error response message is a summary response, with a *CompCode* field of MQCC_FAILED, and a *Reason* field of MQRCCF_COMMAND_FAILED. This message has no parameter structure following the header.

Data response

This response consists of an OK response (as described earlier) to an inquire command. The OK response is followed by additional structures containing the requested data as described in Definitions of the Programmable Command Formats.

Applications must not depend upon these additional parameter structures being returned in any particular order.

Authority checking for PCF commands

When a PCF command is processed, the *UserIdentifier* from the message descriptor in the command message is used for the required WebSphere MQ object authority checks. Authority checking is implemented differently on each platform as described in this topic.

The checks are performed on the system on which the command is being processed; therefore this user ID must exist on the target system and have the required authorities to process the command. If the message has come from a remote system, one way of achieving the ID existing on the target system is to have a matching user ID on both the local and remote systems.

IBM WebSphere MQ for Windows, UNIX and Linux systems

Windows UNIX Linux

In order to process any PCF command, the user ID must have *dsp* authority for the queue manager object on the target system. In addition, WebSphere MQ object authority checks are performed for certain PCF commands, as shown in Table 1 on page 16.

To process any of the following commands the user ID must belong to group *mqm*.

Note: For Windows **only**, the user ID can belong to group *Administrators* or group *mqm*.

- Change Channel
- Copy Channel
- Create Channel
- Delete Channel
- Ping Channel
- Reset Channel
- Start Channel
- Stop Channel
- Start Channel Initiator
- Start Channel Listener
- Resolve Channel
- Reset Cluster
- Refresh Cluster
- Suspend Queue Manager
- Resume Queue Manager

WebSphere MQ for HP Integrity NonStop Server

In order to process any PCF command, the user ID must have *dsp* authority for the queue manager object on the target system. In addition, IBM WebSphere MQ object authority checks are performed for certain PCF commands, as shown in Table 1 on page 16.

To process any of the following commands the user ID must belong to group *mqm*:

- Change Channel
- Copy Channel
- Create Channel
- Delete Channel
- Ping Channel
- Reset Channel
- Start Channel
- Stop Channel

- Start Channel Initiator
- Start Channel Listener
- Resolve Channel
- Reset Cluster
- Refresh Cluster
- Suspend Queue Manager
- Resume Queue Manager

WebSphere MQ Object authorities

Table 1. Windows, HP Integrity NonStop Server, UNIX and Linux systems - object authorities

Command	**WebSphere MQ object authority**	**Class authority (for object type)**
Change Authentication Information	dsp and chg	n/a
Change Channel	dsp and chg	n/a
Change Channel Listener	dsp and chg	n/a
Change Client Connection Channel	dsp and chg	n/a
Change Namelist	dsp and chg	n/a
Change Process	dsp and chg	n/a
Change Queue	dsp and chg	n/a
Change Queue Manager	chg *see Note 3 and Note 5*	n/a
Change Service	dsp and chg	n/a
Clear Queue	clr	n/a
Copy Authentication Information	dsp	crt
Copy Authentication Information (Replace) *see Note 1*	*from:* dsp *to:* chg	crt
Copy Channel	dsp	crt
Copy Channel (Replace) *see Note 1*	*from:* dsp *to:* chg	crt
Copy Channel Listener	dsp	crt
Copy Channel Listener (Replace) *see Note 1*	*from:* dsp *to:* chg	crt
Copy Client Connection Channel	dsp	crt
Copy Client Connection Channel (Replace) *see Note 1*	*from:* dsp *to:* chg	crt
Copy Namelist	dsp	crt
Copy Namelist (Replace) *see Note 1*	*from:* dsp *to:* dsp and chg	crt
Copy Process	dsp	crt

Command	WebSphere MQ object authority	Class authority (for object type)
Copy Process (Replace) *see Note 1*	*from:* dsp *to:* chg	crt
Copy Queue	dsp	crt
Copy Queue (Replace) *see Note 1*	*from:* dsp *to:* dsp and chg	crt
Create Authentication Information	*(system default authentication information)* dsp	crt
Create Authentication Information (Replace) *see Note 1*	*(system default authentication information)* dsp *to:* chg	crt
Create Channel	*(system default channel)* dsp	crt
Create Channel (Replace) *see Note 1*	*(system default channel)* dsp *to:* chg	crt
Create Channel Listener	*(system default listener)* dsp	crt
Create Channel Listener (Replace) *see Note 1*	*(system default listener)* dsp *to:* chg	crt
Create Client Connection Channel	*(system default channel)* dsp	crt
Create Client Connection Channel (Replace) *see Note 1*	*(system default channel)* dsp *to:* chg	crt
Create Namelist	*(system default namelist)* dsp	crt
Create Namelist (Replace) *see Note 1*	*(system default namelist)* dsp *to:* dsp and chg	crt
Create Process	*(system default process)* dsp	crt
Create Process (Replace) *see Note 1*	*(system default process)* dsp *to:* chg	crt
Create Queue	*(system default queue)* dsp	crt
Create Queue (Replace) *see Note 1*	*(system default queue)* dsp *to:* dsp and chg	crt
Create Service	*(system default queue)* dsp	crt
Create Service (Replace) *see Note 1*	*(system default queue)* dsp *to:* chg	crt
Delete Authentication Information	dsp and dlt	n/a
Delete Authority Record	*(queue manager object)* chg *see Note 4*	*see Note 4*
Delete Channel	dsp and dlt	n/a
Delete Channel Listener	dsp and dlt	n/a
Delete Client Connection Channel	dsp and dlt	n/a
Delete Namelist	dsp and dlt	n/a
Delete Process	dsp and dlt	n/a
Delete Queue	dsp and dlt	n/a

Table 1. Windows, HP Integrity NonStop Server, UNIX and Linux systems - object authorities (continued)

Table 1. Windows, HP Integrity NonStop Server, UNIX and Linux systems - object authorities (continued)

Command	**WebSphere MQ object authority**	**Class authority (for object type)**
Delete Service	dsp and dlt	n/a
Inquire Authentication Information	dsp	n/a
Inquire Authority Records	*see Note 4*	*see Note 4*
Inquire Channel	dsp	n/a
Inquire Channel Listener	dsp	n/a
Inquire Channel Status (for **ChannelType** MQCHT_CLSSDR)	inq	n/a
Inquire Client Connection Channel	dsp	n/a
Inquire Namelist	dsp	n/a
Inquire Process	dsp	n/a
Inquire Queue	dsp	n/a
Inquire Queue Manager	*see note 3*	n/a
Inquire Queue Status	dsp	n/a
Inquire Service	dsp	n/a
Ping Channel	ctrl	n/a
Ping Queue Manager	*see note 3*	n/a
Refresh Queue Manager	(queue manager object) chg	n/a
Refresh Security (for **SecurityType** MQSECTYPE_SSL)	(queue manager object) chg	n/a
Reset Channel	ctrlx	n/a
Reset Queue Manager	(queue manager object) chg	n/a
Reset Queue Statistics	dsp and chg	n/a
Resolve Channel	ctrlx	n/a
Set Authority Record	*(queue manager object)* chg *see Note 4*	*see Note 4*
Start Channel	ctrl	n/a
Stop Channel	ctrl	n/a
Stop Connection	(queue manager object) chg	n/a
Start Listener	ctrl	n/a
Stop Listener	ctrl	n/a
Start Service	ctrl	n/a
Stop Service	ctrl	n/a
Escape	*see Note 2*	*see Note 2*

Notes:

1. This command applies if the object to be replaced does exist, otherwise the authority check is as for Create, or Copy without Replace.
2. The required authority is determined by the MQSC command defined by the escape text, and it is equivalent to one of the previous commands.
3. In order to process any PCF command, the user ID must have dsp authority for the queue manager object on the target system.
4. This PCF command is authorized unless the command server has been started with the -a parameter. By default the command server starts when the Queue Manager is started, and without the -a parameter. See the System Administration Guide for further information.
5. Granting a user ID *chg* authority for a queue manager gives the ability to set authority records for all groups and users. Do not grant this authority to ordinary users or applications.

WebSphere MQ also supplies some channel security exit points so that you can supply your own user exit programs for security checking. Details are given in Displaying a channel manual.

Using the MQAI to simplify the use of PCFs

The MQAI is an administration interface to WebSphere MQ that is available on the AIX, HP-UX, IBM i, Linux, Solaris, and Windows platforms.

The MQAI performs administration tasks on a queue manager through the use of *data bags*. Data bags allow you to handle properties (or parameters) of objects in a way that is easier than using PCFs.

Use the MQAI in the following ways:

To simplify the use of PCF messages

The MQAI is an easy way to administer WebSphere MQ; you do not have to write your own PCF messages, avoiding the problems associated with complex data structures.

To pass parameters in programs written using MQI calls, the PCF message must contain the command and details of the string or integer data. To do this, you need several statements in your program for every structure, and memory space must be allocated. This task can be long and laborious.

Programs written using the MQAI pass parameters into the appropriate data bag and you need only one statement for each structure. The use of MQAI data bags removes the need for you to handle arrays and allocate storage, and provides some degree of isolation from the details of the PCF.

To handle error conditions more easily

It is difficult to get return codes back from PCF commands, but the MQAI makes it easier for the program to handle error conditions.

After you have created and populated your data bag, you can send an administration command message to the command server of a queue manager, using the `mqExecute` call, which waits for any response messages. The `mqExecute` call handles the exchange with the command server and returns responses in a *response bag*.

For more information about the MQAI, see "Introduction to the IBM WebSphere MQ Administration Interface (MQAI)" on page 19.

Introduction to the IBM WebSphere MQ Administration Interface (MQAI)

The IBM WebSphere MQ Administration Interface (MQAI) is a programming interface to IBM WebSphere MQ. It performs administration tasks on an IBM WebSphere MQ queue manager using data bags to handle properties (or parameters) of objects in a way that is easier than using Programmable Command Formats (PCFs).

MQAI concepts and terminology

The MQAI is a programming interface to WebSphere MQ, using the C language and also Visual Basic for Windows. It is available on platforms other than z/OS.

It performs administration tasks on a WebSphere MQ queue manager using data bags. Data bags allow you to handle properties (or parameters) of objects in a way that is easier than using the other administration interface, Programmable Command Formats (PCFs). The MQAI offers easier manipulation of PCFs than using the MQGET and MQPUT calls.

For more information about data bags, see "Data bags" on page 46. For more information about PCFs, see "Introduction to Programmable Command Formats" on page 9

Use of the MQAI

You can use the MQAI to:

- Simplify the use of PCF messages. The MQAI is an easy way to administer WebSphere MQ; you do not have to write your own PCF messages and thus avoid the problems associated with complex data structures.
- Handle error conditions more easily. It is difficult to get return codes back from the WebSphere MQ script (MQSC) commands, but the MQAI makes it easier for the program to handle error conditions.
- Exchange data between applications. The application data is sent in PCF format and packed and unpacked by the MQAI. If your message data consists of integers and character strings, you can use the MQAI to take advantage of WebSphere MQ built-in data conversion for PCF data. This avoids the need to write data-conversion exits. For more information on using MQAI to administer WebSphere MQ and to exchange data between applications, see "Using the MQAI to simplify the use of PCFs" on page 19.

Examples of using the MQAI

The list shown gives some example programs that demonstrate the use of MQAI. The samples perform the following tasks:

1. Create a local queue. "Sample C program for creating a local queue (amqsaicq.c)" on page 21
2. Display events on the screen using a simple event monitor. "Sample C program for displaying events using an event monitor (amqsaiem.c)" on page 24
3. Print a list of all local queues and their current depths. "Sample C program for inquiring about queues and printing information (amqsailq.c)" on page 37
4. Print a list of all channels and their types. "Sample C program for inquiring about channel objects (amqsaicl.c)" on page 31

Building your MQAI application

To build your application using the MQAI, you link to the same libraries as you do for WebSphere MQ. For information on how to build your WebSphere MQ applications, see Building a WebSphere MQ application .

Hints and tips for configuring WebSphere MQ using MQAI

The MQAI uses PCF messages to send administration commands to the command server rather than dealing directly with the command server itself. Tips for configuring WebSphere MQ using the MQAI can be found in "Hints and tips for configuring IBM WebSphere MQ" on page 41

IBM WebSphere MQ Administration Interface (MQAI)

IBM WebSphere MQ for Windows, AIX, Linux, HP-UX, and Solaris support the IBM WebSphere MQ Administration Interface (MQAI). The MQAI is a programming interface to IBM WebSphere MQ that gives you an alternative to the MQI, for sending and receiving PCFs.

The MQAI uses *data bags* which allow you to handle properties (or parameters) of objects more easily than using PCFs directly by way of the MQAI.

The MQAI provides easier programming access to PCF messages by passing parameters into the data bag, so that only one statement is required for each structure. This access removes the need for the programmer to handle arrays and allocate storage, and provides some isolation from the details of PCF.

The MQAI administers WebSphere MQ by sending PCF messages to the command server and waiting for a response.

The MQAI is described in the second section of this manual. See the Using Java documentation for a description of a component object model interface to the MQAI.

Sample C program for creating a local queue (amqsaicq.c)

The sample C program `amqsaicq.c` creates a local queue using the MQAI.

```
/******************************************************************************/
/*                                                                            */
/* Program name: AMQSAICQ.C                                                   */
/*                                                                            */
/* Description:  Sample C program to create a local queue using the           */
/*               WebSphere MQ Administration Interface (MQAI).                */
/*                                                                            */
/* Statement:    Licensed Materials - Property of IBM                         */
/*                                                                            */
/*               84H2000, 5765-B73                                            */
/*               84H2001, 5639-B42                                            */
/*               84H2002, 5765-B74                                            */
/*               84H2003, 5765-B75                                            */
/*               84H2004, 5639-B43                                            */
/*                                                                            */
/*               (C) Copyright IBM Corp. 1999, 2025.                          */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Function:                                                                  */
/*    AMQSAICQ is a sample C program that creates a local queue and is an     */
/*    example of the use of the mqExecute call.                               */
/*                                                                            */
/*     - The name of the queue to be created is a parameter to the program.   */
/*                                                                            */
/*     - A PCF command is built by placing items into an MQAI bag.            */
/*       These are:-                                                          */
/*            - The name of the queue                                         */
/*            - The type of queue required, which, in this case, is local.    */
/*                                                                            */
/*     - The mqExecute call is executed with the command MQCMD_CREATE_Q.      */
/*       The call generates the correct PCF structure.                        */
/*       The call receives the reply from the command server and formats into */
/*       the response bag.                                                    */
/*                                                                            */
/*     - The completion code from the mqExecute call is checked and if there  */
/*       is a failure from the command server then the code returned by the   */
/*       command server is retrieved from the system bag that is              */
/*       embedded in the response bag to the mqExecute call.                  */
/*                                                                            */
/* Note: The command server must be running.                                  */
/*                                                                            */
/*                                                                            */

/******************************************************************************/
/*                                                                            */
/* AMQSAICQ has 2 parameters - the name of the local queue to be created      */
/*                           - the queue manager name (optional)              */
/*                                                                            */
/******************************************************************************/
/******************************************************************************/
/* Includes                                                                   */
/******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include <cmqc.h>                          /* MQI                             */
#include <cmqcfc.h>                        /* PCF                             */
#include <cmqbc.h>                         /* MQAI                            */

void CheckCallResult(MQCHAR *, MQLONG , MQLONG );
void CreateLocalQueue(MQHCONN, MQCHAR *);

int main(int argc, char *argv[])
{
   MQHCONN hConn;                          /* handle to WebSphere MQ connection   */
   MQCHAR QMName[MQ_Q_MGR_NAME_LENGTH+1]=""; /* default QMgr name   */
   MQLONG connReason;                      /* MQCONN reason code              */
   MQLONG compCode;                        /* completion code                 */
   MQLONG reason;                          /* reason code                     */

   /***************************************************************************/
   /* First check the required parameters                                     */
   /***************************************************************************/
   printf("Sample Program to Create a Local Queue\n");
   if (argc < 2)
   {
     printf("Required parameter missing - local queue name\n");
     exit(99);
   }

   /***************************************************************************/
   /* Connect to the queue manager                                            */
   /***************************************************************************/
   if (argc > 2)
      strncpy(QMName, argv[2], (size_t)MQ_Q_MGR_NAME_LENGTH);
      MQCONN(QMName, &hConn, &compCode, &connReason);

/******************************************************************************/
/* Report reason and stop if connection failed                               */
/******************************************************************************/
   if (compCode == MQCC_FAILED)
   {
      CheckCallResult("MQCONN", compCode, connReason);
      exit( (int)connReason);
   }


/******************************************************************************/
/* Call the routine to create a local queue, passing the handle to the        */
/* queue manager and also passing the name of the queue to be created.        */
/******************************************************************************/
   CreateLocalQueue(hConn, argv[1]);

   /***************************************************************************/
   /* Disconnect from the queue manager if not already connected              */
   /***************************************************************************/
   if (connReason != MQRC_ALREADY_CONNECTED)
   {
      MQDISC(&hConn, &compCode, &reason);
      CheckCallResult("MQDISC", compCode, reason);
   }
   return 0;

}
/******************************************************************************/
/*                                                                            */
/* Function:    CreateLocalQueue                                              */
/* Description: Create a local queue by sending a PCF command to the command  */
/*              server.                                                       */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Input Parameters:  Handle to the queue manager                             */
/*                    Name of the queue to be created                         */
/*                                                                            */
/* Output Parameters: None                                                    */
/*                                                                            */
/* Logic: The mqExecute call is executed with the command MQCMD_CREATE_Q.     */
/*        The call generates the correct PCF structure.                       */
/*        The default options to the call are used so that the command is sent*/
/*        to the SYSTEM.ADMIN.COMMAND.QUEUE.                                  */
/*        The reply from the command server is placed on a temporary dynamic  */
/*        queue.                                                              */
/*        The reply is read from the temporary queue and formatted into the   */
/*        response bag.                                                       */
/*                                                                            */
/*        The completion code from the mqExecute call is checked and if there */
/*        is a failure from the command server then the code returned by the  */
/*        command server is retrieved from the system bag that is             */
/*        embedded in the response bag to the mqExecute call.                 */
/*                                                                            */
/******************************************************************************/
```

```
void CreateLocalQueue(MQHCONN hConn, MQCHAR *qName)
{
   MQLONG reason;                        /* reason code                   */
   MQLONG compCode;                      /* completion code               */
   MQHBAG commandBag = MQHB_UNUSABLE_HBAG; /* command bag for mqExecute     */
   MQHBAG responseBag = MQHB_UNUSABLE_HBAG;/* response bag for mqExecute    */
   MQHBAG resultBag;                     /* result bag from mqExecute     */
   MQLONG mqExecuteCC;                   /* mqExecute completion code     */
   MQLONG mqExecuteRC;                   /* mqExecute reason code         */

   printf("\nCreating Local Queue %s\n\n", qName);


   /***************************************************************************/
   /* Create a command Bag for the mqExecute call. Exit the function if the   */
   /* create fails.                                                           */
   /***************************************************************************/
   mqCreateBag(MQCBO_ADMIN_BAG, &commandBag, &compCode, &reason);
   CheckCallResult("Create the command bag", compCode, reason);
   if (compCode !=MQCC_OK)
      return;

   /***************************************************************************/
   /* Create a response Bag for the mqExecute call, exit the function if the  */
   /* create fails.                                                           */
   /***************************************************************************/
   mqCreateBag(MQCBO_ADMIN_BAG, &responseBag, &compCode, &reason);
   CheckCallResult("Create the response bag", compCode, reason);
   if (compCode !=MQCC_OK)
      return;

   /***************************************************************************/
   /* Put the name of the queue to be created into the command bag. This will */
   /* be used by the mqExecute call.                                          */
   /***************************************************************************/
   mqAddString(commandBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED, qName, &compCode,
             &reason);
   CheckCallResult("Add q name to command bag", compCode, reason);

   /***************************************************************************/
   /* Put queue type of local into the command bag. This will be used by the  */
   /* mqExecute call.                                                         */
   /***************************************************************************/
   mqAddInteger(commandBag, MQIA_Q_TYPE, MQQT_LOCAL, &compCode, &reason);
   CheckCallResult("Add q type to command bag", compCode, reason);

   /***************************************************************************/
   /* Send the command to create the required local queue.                    */
   /* The mqExecute call will create the PCF structure required, send it to   */
   /* the command server and receive the reply from the command server into   */
   /* the response bag.                                                       */
   /***************************************************************************/
   mqExecute(hConn,                  /* WebSphere MQ connection handle           */
             MQCMD_CREATE_Q,         /* Command to be executed               */
             MQHB_NONE,              /* No options bag                       */
             commandBag,             /* Handle to bag containing commands    */
             responseBag,            /* Handle to bag to receive the response*/
             MQHO_NONE,              /* Put msg on SYSTEM.ADMIN.COMMAND.QUEUE*/
             MQHO_NONE,              /* Create a dynamic q for the response  */
             &compCode,              /* Completion code from the mqExecute   */
             &reason);               /* Reason code from mqExecute call      */

   if (reason == MQRC_CMD_SERVER_NOT_AVAILABLE)
   {
      printf("Please start the command server: <strmqcsv QMgrName>\n")
      MQDISC(&hConn, &compCode, &reason);
      CheckCallResult("MQDISC", compCode, reason);
      exit(98);
   }


   /***************************************************************************/
   /* Check the result from mqExecute call and find the error if it failed.   */
   /***************************************************************************/
   if ( compCode == MQCC_OK )
      printf("Local queue %s successfully created\n", qName);
   else
   {
      printf("Creation of local queue %s failed: Completion Code = %d
              qName, compCode, reason);
      if (reason == MQRCCF_COMMAND_FAILED)
      {
         /********************************************************************/
         /* Get the system bag handle out of the mqExecute response bag.     */
         /* This bag contains the reason from the command server why the     */
         /* command failed.                                                  */
         /********************************************************************/
         mqInquireBag(responseBag, MQHA_BAG_HANDLE, 0, &resultBag, &compCode,
                      &reason);
         CheckCallResult("Get the result bag handle", compCode, reason);

         /********************************************************************/
         /* Get the completion code and reason code, returned by the command  */
         /* server, from the embedded error bag.                             */
         /********************************************************************/
         mqInquireInteger(resultBag, MQIASY_COMP_CODE, MQIND_NONE, &mqExecuteCC,
                          &compCode, &reason);
         CheckCallResult("Get the completion code from the result bag",
                          compCode, reason);
         mqInquireInteger(resultBag, MQIASY_REASON, MQIND_NONE, &mqExecuteRC,
                          &compCode, &reason);
         CheckCallResult("Get the reason code from the result bag", compCode,
                          reason);
         printf("Error returned by the command server: Completion code = %d :
                 Reason = %d\n", mqExecuteCC, mqExecuteRC);
      }
   }
   /***************************************************************************/
   /* Delete the command bag if successfully created.                         */
   /***************************************************************************/
   if (commandBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&commandBag, &compCode, &reason);
      CheckCallResult("Delete the command bag", compCode, reason);
   }

   /***************************************************************************/
   /* Delete the response bag if successfully created.                        */
   /***************************************************************************/
   if (responseBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&responseBag, &compCode, &reason);
      CheckCallResult("Delete the response bag", compCode, reason);
   }
} /* end of CreateLocalQueue */


/******************************************************************************/
/*                                                                            */
/* Function: CheckCallResult                                                  */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Input Parameters:  Description of call                                     */
/*                    Completion code                                         */
/*                    Reason code                                             */
/*                                                                            */
/* Output Parameters: None                                                    */
/*                                                                            */
/* Logic: Display the description of the call, the completion code and the    */
/*        reason code if the completion code is not successful                */
/*                                                                            */
/******************************************************************************/
void  CheckCallResult(char *callText, MQLONG cc, MQLONG rc)
{
   if (cc != MQCC_OK)
         printf("%s failed: Completion Code = %d :
                 Reason = %d\n", callText, cc, rc);

}
```

Sample C program for displaying events using an event monitor (amqsaiem.c)

The sample C program `amqsaiem.c` demonstrates a basic event monitor using the MQAI.

```
******************************************************************************/
/*                                                                            */
/* Program name: AMQSAIEM.C                                                   */
/*                                                                            */
/* Description:  Sample C program to demonstrate a basic event monitor        */
/*                using the WebSphere MQ Admin Interface (MQAI).               */
/* Licensed Materials - Property of IBM                                        */
/*                                                                             */
/* 63H9336                                                                     */
/* (c) Copyright IBM Corp. 1999, 2025. All Rights Reserved.                    */
/*                                                                             */
/* US Government Users Restricted Rights - Use, duplication or                 */
/* disclosure restricted by GSA ADP Schedule Contract with                     */
/* IBM Corp.                                                                   */
/******************************************************************************/
/*                                                                             */
/* Function:                                                                   */
/*    AMQSAIEM is a sample C program that demonstrates how to write a simple  */
/*    event monitor using the mqGetBag call and other MQAI calls.              */
/*                                                                             */
/*    The name of the event queue to be monitored is passed as a parameter    */
/*    to the program. This would usually be one of the system event queues:-  */
/*            SYSTEM.ADMIN.QMGR.EVENT        Queue Manager events              */
/*            SYSTEM.ADMIN.PERFM.EVENT       Performance events                */
/*            SYSTEM.ADMIN.CHANNEL.EVENT     Channel events                    */
/*            SYSTEM.ADMIN.LOGGER.EVENT      Logger events                     */
/*                                                                             */
/*    To monitor the queue manager event queue or the performance event queue,*/
/*    the attributes of the queue manager needs to be changed to enable       */
/*    these events. For more information about this, see Part 1 of the        */
/*    Programmable System Management book. The queue manager attributes can   */
/*    be changed using either MQSC commands or the MQAI interface.            */
/*    Channel events are enabled by default.                                  */
/*                                                                             */
/* Program logic                                                               */
/*    Connect to the Queue Manager.                                            */
/*    Open the requested event queue with a wait interval of 30 seconds.      */
/*    Wait for a message, and when it arrives get the message from the queue  */
/*    and format it into an MQAI bag using the mqGetBag call.                  */
/*    There are many types of event messages and it is beyond the scope of    */
/*    this sample to program for all event messages. Instead the program      */
/*    prints out the contents of the formatted bag.                           */
/*    Loop around to wait for another message until either there is an error  */
/*    or the wait interval of 30 seconds is reached.                          */
/*                                                                             */
/******************************************************************************/
/*                                                                             */
/* AMQSAIEM has 2 parameters - the name of the event queue to be monitored    */
/*                           - the queue manager name (optional)              */
/*                                                                             */
/*****************************************************************************

/******************************************************************************/
/* Includes                                                                    */
/******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include <cmqc.h>                         /* MQI                               */
#include <cmqcfc.h>                       /* PCF                               */
#include <cmqbc.h>                        /* MQAI                              */

/******************************************************************************/
/* Macros                                                                      */
/******************************************************************************/
#if MQAT_DEFAULT == MQAT_WINDOWS_NT
  #define Int64 "I64"
#elif defined(MQ_64_BIT)
  #define Int64 "l"
#else
  #define Int64 "ll"
#endif

/******************************************************************************/
/* Function prototypes                                                         */
/******************************************************************************/
void CheckCallResult(MQCHAR *, MQLONG , MQLONG);
void GetQEvents(MQHCONN, MQCHAR *);
int PrintBag(MQHBAG);
int PrintBagContents(MQHBAG, int);

/******************************************************************************/
/* Function: main                                                              */
/******************************************************************************/
int main(int argc, char *argv[])
```

```
{
   MQHCONN hConn;                          /* handle to connection            */
   MQCHAR QMName[MQ_Q_MGR_NAME_LENGTH+1]=""; /* default QM name               */
   MQLONG reason;                          /* reason code                     */
   MQLONG connReason;                      /* MQCONN reason code              */
   MQLONG compCode;                        /* completion code                 */

   /*************************************************************************/
   /* First check the required parameters                                   */
   /*************************************************************************/
   printf("Sample Event Monitor (times out after 30 secs)\n");
   if (argc < 2)
   {
     printf("Required parameter missing - event queue to be monitored\n");
     exit(99);
   }

   /************************************************************************/
   /* Connect to the queue manager                                         */
   /************************************************************************/
   if (argc > 2)
     strncpy(QMName, argv[2], (size_t)MQ_Q_MGR_NAME_LENGTH);
   MQCONN(QMName, &hConn, &compCode, &connReason);
   /*************************************************************************/
   /* Report the reason and stop if the connection failed                   */
   /*************************************************************************/
   if (compCode == MQCC_FAILED)
   {
      CheckCallResult("MQCONN", compCode, connReason);
      exit( (int)connReason);
   }

   /*************************************************************************/
   /* Call the routine to open the event queue and format any event messages  */
   /* read from the queue.                                                  */
   /*************************************************************************/
   GetQEvents(hConn, argv[1]);

   /*************************************************************************/
   /* Disconnect from the queue manager if not already connected            */
   /*************************************************************************/
   if (connReason != MQRC_ALREADY_CONNECTED)
   {
      MQDISC(&hConn, &compCode, &reason);
      CheckCallResult("MQDISC", compCode, reason);
   }

   return 0;

}

/****************************************************************************/
/*                                                                          */
/* Function: CheckCallResult                                                */
/*                                                                          */
/****************************************************************************/
/*                                                                          */
/* Input Parameters:  Description of call                                   */
/*                    Completion code                                       */
/*                    Reason code                                           */
/*                                                                          */
/* Output Parameters: None                                                  */
/*                                                                          */
/* Logic: Display the description of the call, the completion code and the  */
/*        reason code if the completion code is not successful              */
/*                                                                          */
/****************************************************************************/
void  CheckCallResult(char *callText, MQLONG cc, MQLONG rc)
{
   if (cc != MQCC_OK)
         printf("%s failed: Completion Code = %d : Reason = %d\n",
                 callText, cc, rc);

}

/****************************************************************************/
/*                                                                          */
/* Function: GetQEvents                                                     */
/*                                                                          */
/****************************************************************************/
/*                                                                          */
/* Input Parameters:  Handle to the queue manager                           */
/*                    Name of the event queue to be monitored               */
/*                                                                          */
/* Output Parameters: None                                                  */
/*
/* Logic:   Open the event queue.                                           */
/*          Get a message off the event queue and format the message into   */
/*          a bag.                                                          */
/*          A real event monitor would need to be programmed to deal with   */
/*          each type of event that it receives from the queue. This is     */
/*          outside the scope of this sample, so instead, the contents of   */
/*          the bag are printed.                                            */
/*          The program waits for 30 seconds for an event message and then  */
/*          terminates if no more messages are available.                   */
/*                                                                          */
/****************************************************************************/
void GetQEvents(MQHCONN hConn, MQCHAR *qName)
{
   MQLONG openReason;                      /* MQOPEN reason code             */
   MQLONG reason;                          /* reason code                    */
   MQLONG compCode;                        /* completion code                */
   MQHOBJ eventQueue;                      /* handle to event queue          */

   MQHBAG eventBag = MQHB_UNUSABLE_HBAG;   /* event bag to receive event msg */
   MQOD   od = {MQOD_DEFAULT};             /* Object Descriptor              */
   MQMD   md = {MQMD_DEFAULT};             /* Message Descriptor             */
   MQGMO  gmo = {MQGMO_DEFAULT};           /* get message options            */
   MQLONG bQueueOK = 1;                    /* keep reading msgs while true   */

   /*************************************************************************/
   /* Create an Event Bag in which to receive the event.                    */
   /* Exit the function if the create fails.                                */
   /*************************************************************************/
   mqCreateBag(MQCBO_USER_BAG, &eventBag, &compCode, &reason);
   CheckCallResult("Create event bag", compCode, reason);
   if (compCode !=MQCC_OK)
      return;

   /*************************************************************************/
   /* Open the event queue chosen by the user                               */
   /*************************************************************************/
   strncpy(od.ObjectName, qName, (size_t)MQ_Q_NAME_LENGTH);
   MQOPEN(hConn, &od, MQOO_INPUT_AS_Q_DEF+MQOO_FAIL_IF_QUIESCING, &eventQueue,
          &compCode, &openReason);
   CheckCallResult("Open event queue", compCode, openReason);

   /*************************************************************************/
   /* Set the GMO options to control the action of the get message from the */
   /* queue.                                                                */
   /*************************************************************************/
   gmo.WaitInterval = 30000;              /* 30 second wait for message      */
   gmo.Options = MQGMO_WAIT + MQGMO_FAIL_IF_QUIESCING + MQGMO_CONVERT;
   gmo.Version = MQGMO_VERSION_2;         /* Avoid need to reset Message ID  */
   gmo.MatchOptions = MQMO_NONE;          /* and Correlation ID after every  */
                                   /* mqGetBag
   /*************************************************************************/
   /* If open fails, we cannot access the queue and must stop the monitor.  */
   /*************************************************************************/
   if (compCode != MQCC_OK)
      bQueueOK = 0;

   /*************************************************************************/
   /* Main loop to get an event message when it arrives                     */
   /*************************************************************************/
   while (bQueueOK)
   {
     printf("\nWaiting for an event\n");

     /***********************************************************************/
     /* Get the message from the event queue and convert it into the event  */
     /* bag.                                                                */
     /***********************************************************************/
     mqGetBag(hConn, eventQueue, &md, &gmo, eventBag, &compCode, &reason);

     /***********************************************************************/
     /* If get fails, we cannot access the queue and must stop the monitor. */
     /***********************************************************************/
     if (compCode != MQCC_OK)
     {
        bQueueOK = 0;

        /*******************************************************************/
        /* If get fails because no message available then we have timed out, */
```

```
         /* so report this, otherwise report an error.                      */
         /******************************************************************/
         if (reason == MQRC_NO_MSG_AVAILABLE)
         {
            printf("No more messages\n");
         }
         else
         {
            CheckCallResult("Get bag", compCode, reason);
         }
      }

      /*********************************************************************/
      /* Event message read - Print the contents of the event bag          */
      /*********************************************************************/
      else
      {
        if ( PrintBag(eventBag) )
            printf("\nError found while printing bag contents\n");

      }  /* end of msg found */
    } /* end of main loop */
    /***********************************************************************/
    /* Close the event queue if successfully opened                        */
    /***********************************************************************/
    if (openReason == MQRC_NONE)
    {
       MQCLOSE(hConn, &eventQueue, MQCO_NONE, &compCode, &reason);
       CheckCallResult("Close event queue", compCode, reason);
    }

    /***********************************************************************/
    /* Delete the event bag if successfully created.                       */
    /***********************************************************************/
    if (eventBag != MQHB_UNUSABLE_HBAG)
    {
       mqDeleteBag(&eventBag, &compCode, &reason);
       CheckCallResult("Delete the event bag", compCode, reason);
    }

 } /* end of GetQEvents */

 /**************************************************************************/
 /*                                                                        */
 /* Function: PrintBag                                                     */
 /*                                                                        */
 /**************************************************************************/
 /*                                                                        */
 /* Input Parameters:  Bag Handle                                          */
 /*                                                                        */
 /* Output Parameters: None                                                */
 /*                                                                        */
 /* Returns:           Number of errors found                              */
 /*                                                                        */
 /* Logic: Calls PrintBagContents to display the contents of the bag.      */
 /*                                                                        */
 /*************************************************************************

 int PrintBag(MQHBAG dataBag)
 {
     int errors;

     printf("\n");
     errors = PrintBagContents(dataBag, 0);
     printf("\n");

     return errors;
 }

 /**************************************************************************/
 /*                                                                        */
 /* Function: PrintBagContents                                             */
 /*                                                                        */
 /**************************************************************************/
 /*                                                                        */
 /* Input Parameters:  Bag Handle                                          */
 /*                    Indentation level of bag                            */
 /*                                                                        */
 /* Output Parameters: None                                                */
 /*                                                                        */
 /* Returns:           Number of errors found                              */
 /*                                                                        */
```

```
/* Logic: Count the number of items in the bag                                */
/*        Obtain selector and item type for each item in the bag.             */
/*        Obtain the value of the item depending on item type and display the */
/*        index of the item, the selector and the value.                      */
/*        If the item is an embedded bag handle then call this function again */
/*        to print the contents of the embedded bag increasing the            */
/*        indentation level.                                                  */
/*                                                                            */
/******************************************************************************/
int PrintBagContents(MQHBAG dataBag, int indent)
{
   /***************************************************************************/
   /* Definitions                                                             */
   /***************************************************************************/
   #define LENGTH 500                       /* Max length of string to be read*/
   #define INDENT 4                         /* Number of spaces to indent     */
                                            /* embedded bag display           */


   /***************************************************************************/
   /* Variables                                                               */
   /***************************************************************************/
   MQLONG  itemCount;                       /* Number of items in the bag     */
   MQLONG  itemType;                        /* Type of the item               */
   int     i;                               /* Index of item in the bag       */
   MQCHAR  stringVal[LENGTH+1];             /* Value if item is a string      */
   MQBYTE  byteStringVal[LENGTH];           /* Value if item is a byte string */
   MQLONG  stringLength;                    /* Length of string value         */
   MQLONG  ccsid;                           /* CCSID of string value          */
   MQINT32 iValue;                          /* Value if item is an integer    */
   MQINT64 i64Value;                        /* Value if item is a 64-bit      */
                                            /* integer                        */
   MQLONG  selector;                        /* Selector of item               */
   MQHBAG  bagHandle;                       /* Value if item is a bag handle  */
   MQLONG  reason;                          /* reason code                    */
   MQLONG  compCode;                        /* completion code                */
   MQLONG  trimLength;                      /* Length of string to be trimmed */
   int     errors = 0;                      /* Count of errors found          */
   char    blanks[] = "                       "; /* Blank string used to      */
                                                 /* indent display            */

   /***************************************************************************/
   /* Count the number of items in the bag                                    */
   /***************************************************************************/
   mqCountItems(dataBag, MQSEL_ALL_SELECTORS, &itemCount, &compCode, &reason);

   if (compCode != MQCC_OK)
      errors++;
   else
   {
      printf("
      printf("
      printf("
   }

   /***************************************************************************/
   /* If no errors found, display each item in the bag                        */
   /***************************************************************************/
   if (!errors)
   {
      for (i = 0; i < itemCount; i++)
      {

          /*******************************************************************/
          /* First inquire the type of the item for each item in the bag      */
          /*******************************************************************/
          mqInquireItemInfo(dataBag,             /* Bag handle                */
                            MQSEL_ANY_SELECTOR,  /* Item can have any selector*/
                            i,                   /* Index position in the bag */
                            &selector,           /* Actual value of selector  */
                                                 /* returned by call          */
                            &itemType,           /* Actual type of item       */
                                                 /* returned by call          */
                            &compCode,           /* Completion code           */
                            &reason);            /* Reason Code               */

          if (compCode != MQCC_OK)
             errors++;

          switch(itemType)
          {
          case MQITEM_INTEGER:
            /***********************************************************/
            /* Item is an integer. Find its value and display its index,   */
            /* selector and value.                                         */
            /***********************************************************/
            mqInquireInteger(dataBag,         /* Bag handle              */
                             MQSEL_ANY_SELECTOR, /* Allow any selector     */
                             i,                /* Index position in the bag */
                             &iValue,          /* Returned integer value
                             &compCode,        /* Completion code         */
                             &reason);         /* Reason Code             */

            if (compCode != MQCC_OK)
               errors++;
            else
               printf("%.*s  %-2d     %-4d     (%d)\n",
                       indent, blanks, i, selector, iValue);
            break

       case MQITEM_INTEGER64:
            /***********************************************************/
            /* Item is a 64-bit integer. Find its value and display its    */
            /* index, selector and value.                                  */
            /***********************************************************/
            mqInquireInteger64(dataBag,       /* Bag handle              */
                               MQSEL_ANY_SELECTOR, /* Allow any selector   */
                               i,              /* Index position in the bag */
                               &i64Value,      /* Returned integer value   */
                               &compCode,      /* Completion code          */
                               &reason);       /* Reason Code              */

            if (compCode != MQCC_OK)
               errors++;
            else
               printf("%.*s  %-2d     %-4d     (%"Int64"d)\n",
                       indent, blanks, i, selector, i64Value);
            break;


      case MQITEM_STRING:
            /***********************************************************/
            /* Item is a string. Obtain the string in a buffer, prepare    */
            /* the string for displaying and display the index, selector,  */
            /* string and Character Set ID.                                */
            /***********************************************************/
            mqInquireString(dataBag,          /* Bag handle              */
                            MQSEL_ANY_SELECTOR, /* Allow any selector      */
                            i,                 /* Index position in the bag */
                            LENGTH,            /* Maximum length of buffer  */
                            stringVal,         /* Buffer to receive string  */
                            &stringLength,     /* Actual length of string   */
                            &ccsid,            /* Coded character set id    */
                            &compCode,         /* Completion code           */
                            &reason);          /* Reason Code               */

            /***********************************************************/
            /* The call can return a warning if the string is too long for */
            /* the output buffer and has been truncated, so only check     */
            /* explicitly for call failure.                                */
            /***********************************************************/
            if (compCode == MQCC_FAILED)
                errors++;
            else
            {
               /********************************************************/
               /* Remove trailing blanks from the string and terminate with*/
               /* a null. First check that the string should not have been */
               /* longer than the maximum buffer size allowed.             */
               /********************************************************/
               if (stringLength > LENGTH)
                  trimLength = LENGTH;
               else
                  trimLength = stringLength;
               mqTrim(trimLength, stringVal, stringVal, &compCode, &reason);
               printf("%.*s  %-2d     %-4d     '%s' %d\n",
                       indent, blanks, i, selector, stringVal, ccsid);
            }
            break;

       case MQITEM_BYTE_STRING:
            /***********************************************************/
            /* Item is a byte string. Obtain the byte string in a buffer,  */
            /* prepare the byte string for displaying and display the      */
```

```
               /* index, selector and string.                                 */
               /***************************************************************/
               mqInquireByteString(dataBag,      /* Bag handle                */
                                   MQSEL_ANY_SELECTOR, /* Allow any selector  */
                                   i,            /* Index position in the bag */
                                   LENGTH,       /* Maximum length of buffer  */
                                   byteStringVal, /* Buffer to receive string */
                                   &stringLength, /* Actual length of string  */
                                   &compCode,    /* Completion code           */
                                   &reason);     /* Reason Code

               /***************************************************************/
               /* The call can return a warning if the string is too long for */
               /* the output buffer and has been truncated, so only check     */
               /* explicitly for call failure.                                */
               /***************************************************************/
               if (compCode == MQCC_FAILED)
                   errors++;
               else
               {
                  printf("%.*s  %-2d     %-4d     X'",
                         indent, blanks, i, selector);

                  for (i = 0 ; i < stringLength ; i++)
                     printf("

                  printf("'\n");
               }
               break;

          case MQITEM_BAG:
               /***************************************************************/
               /* Item is an embedded bag handle, so call the PrintBagContents*/
               /* function again to display the contents.                     */
               /***************************************************************/
               mqInquireBag(dataBag,             /* Bag handle                */
                            MQSEL_ANY_SELECTOR,  /* Allow any selector        */
                            i,                   /* Index position in the bag */
                            &bagHandle,          /* Returned embedded bag hdle*/
                            &compCode,           /* Completion code           */
                            &reason);            /* Reason Code               */

               if (compCode != MQCC_OK)
                  errors++;
               else
               {
                  printf("%.*s  %-2d     %-4d     (%d)\n", indent, blanks, i,
                          selector, bagHandle);
                  if (selector == MQHA_BAG_HANDLE)
                     printf("
                  else
                     printf("
                  PrintBagContents(bagHandle, indent+INDENT);
               }
               break;

          default:
               printf("
          }
      }
   }
   return errors;
}
```

Sample C program for inquiring about channel objects (amqsaicl.c)

The sample C program `amqsaicl.c` inquires channel objects using the MQAI.

```
 /******************************************************************************/
/*                                                                            */
/* Program name: AMQSAICL.C                                                   */
/*                                                                            */
/* Description:  Sample C program to inquire channel objects                  */
/*               using the WebSphere MQ Administration Interface (MQAI)       */
/*                                                                            */
/* <N_OCO_COPYRIGHT>                                                          */
/* Licensed Materials - Property of IBM                                       */
/*                                                                            */
/* 63H9336                                                                    */
/* (c) Copyright IBM Corp. 2008, 2025. All Rights Reserved.                  */
/*                                                                           */
/* US Government Users Restricted Rights - Use, duplication or               */
/* disclosure restricted by GSA ADP Schedule Contract with                   */
/* IBM Corp.                                                                 */
/* <NOC_COPYRIGHT>                                                           */
/*****************************************************************************/
/*                                                                           */
/* Function:                                                                 */
/*    AMQSAICL is a sample C program that demonstrates how to inquire        */
/*    attributes of the local queue manager using the MQAI interface. In     */
/*    particular, it inquires all channels and their types.                  */
/*                                                                           */
/*     - A PCF command is built from items placed into an MQAI administration */
/*       bag.                                                                */
/*       These are:-                                                         */
/*            - The generic channel name "*"                                 */
/*            - The attributes to be inquired. In this sample we just want   */
/*              name and type attributes                                     */
/*                                                                           */
/*     - The mqExecute MQCMD_INQUIRE_CHANNEL call is executed.               */
/*       The call generates the correct PCF structure.                       */
/*       The default options to the call are used so that the command is sent */
/*       to the SYSTEM.ADMIN.COMMAND.QUEUE.                                  */
/*       The reply from the command server is placed on a temporary dynamic  */
/*       queue.                                                              */
/*       The reply from the MQCMD_INQUIRE_CHANNEL is read from the           */
/*       temporary queue and formatted into the response bag.                */
/*                                                                           */
/*     - The completion code from the mqExecute call is checked and if there */
/*       is a failure from the command server, then the code returned by the */
/*       command server is retrieved from the system bag that has been       */
/*       embedded in the response bag to the mqExecute call.                 */
/*                                                                           */
/* Note: The command server must be running.                                 */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/* AMQSAICL has 2 parameter - the queue manager name (optional)              */
/*                          - output file (optional) default varies          */
/*****************************************************************************/

/*****************************************************************************/
/* Includes                                                                  */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#if (MQAT_DEFAULT == MQAT_OS400)
#include <recio.h>
#endif

#include <cmqc.h>                          /* MQI                            */
#include <cmqcfc.h>                        /* PCF                            */
#include <cmqbc.h>                         /* MQAI                           */
#include <cmqxc.h>                         /* MQCD                           */

/*****************************************************************************/
/* Function prototypes                                                       */
/*****************************************************************************/
void CheckCallResult(MQCHAR *, MQLONG , MQLONG);

/*****************************************************************************/
/* DataTypes                                                                 */
/*****************************************************************************/
#if (MQAT_DEFAULT == MQAT_OS400)
typedef _RFILE OUTFILEHDL;
#else
typedef FILE OUTFILEHDL;
#endif

/*****************************************************************************/
/* Constants                                                                 */
/*****************************************************************************/
#if (MQAT_DEFAULT == MQAT_OS400)
const struct
{
  char name[9];
} ChlTypeMap[9] =
{
  "*SDR     ",    /* MQCHT_SENDER    */
  "*SVR     ",    /* MQCHT_SERVER    */
  "*RCVR    ",    /* MQCHT_RECEIVER  */
  "*RQSTR   ",    /* MQCHT_REQUESTER */
  "*ALL     ",    /* MQCHT_ALL       */
  "*CLTCN   ",    /* MQCHT_CLNTCONN  */
  "*SVRCONN ",    /* MQCHT_SVRCONN   */
  "*CLUSRCVR",    /* MQCHT_CLUSRCVR  */
  "*CLUSSDR "     /* MQCHT_CLUSSDR   */
};
#else
const struct
{
  char name[9];
} ChlTypeMap[9] =
{
  "sdr      ",    /* MQCHT_SENDER    */
  "svr      ",    /* MQCHT_SERVER    */
  "rcvr     ",    /* MQCHT_RECEIVER  */
  "rqstr    ",    /* MQCHT_REQUESTER */
  "all      ",    /* MQCHT_ALL       */
  "cltconn  ",    /* MQCHT_CLNTCONN  */
  "svrcn    ",    /* MQCHT_SVRCONN   */
  "clusrcvr ",    /* MQCHT_CLUSRCVR  */
  "clussdr  "     /* MQCHT_CLUSSDR   */
};
#endif

/******************************************************************************/
/* Macros                                                                     */
/******************************************************************************/
#if (MQAT_DEFAULT == MQAT_OS400)
  #define OUTFILE "QTEMP/AMQSAICL(AMQSAICL)"
  #define OPENOUTFILE(hdl, fname) \
    (hdl) = _Ropen((fname),"wr, rtncode=Y");
  #define CLOSEOUTFILE(hdl) \
    _Rclose((hdl));
  #define WRITEOUTFILE(hdl, buf, buflen) \
    _Rwrite((hdl),(buf),(buflen));

#elif (MQAT_DEFAULT == MQAT_UNIX)
  #define OUTFILE "/tmp/amqsaicl.txt"
  #define OPENOUTFILE(hdl, fname) \
    (hdl) = fopen((fname),"w");
  #define CLOSEOUTFILE(hdl) \
    fclose((hdl));
  #define WRITEOUTFILE(hdl, buf, buflen) \
    fwrite((buf),(buflen),1,(hdl)); fflush((hdl));

#else
  #define OUTFILE "amqsaicl.txt"
  #define OPENOUTFILE(fname) \
    fopen((fname),"w");
  #define CLOSEOUTFILE(hdl) \
    fclose((hdl));
  #define WRITEOUTFILE(hdl, buf, buflen) \
    fwrite((buf),(buflen),1,(hdl)); fflush((hdl));

#endif

#define ChlType2String(t) ChlTypeMap[(t)-1].name

/******************************************************************************/
/* Function: main                                                             */
/******************************************************************************/
int main(int argc, char *argv[])
{
   /***************************************************************************/
   /* MQAI variables                                                          */
   /***************************************************************************/
   MQHCONN hConn;                          /* handle to MQ connection         */
   MQCHAR qmName[MQ_Q_MGR_NAME_LENGTH+1]=""; /* default QMgr name             */
   MQLONG reason;                          /* reason code                     */
   MQLONG connReason;                      /* MQCONN reason code              */
   MQLONG compCode;                        /* completion code                 */
   MQHBAG adminBag = MQHB_UNUSABLE_HBAG;   /* admin bag for mqExecute         */
   MQHBAG responseBag = MQHB_UNUSABLE_HBAG;/* response bag for mqExecute      */
   MQHBAG cAttrsBag;                       /* bag containing chl attributes   */
   MQHBAG errorBag;                        /* bag containing cmd server error */
   MQLONG mqExecuteCC;                     /* mqExecute completion code       */
   MQLONG mqExecuteRC;                     /* mqExecute reason code           */
   MQLONG chlNameLength;                   /* Actual length of chl name       */
   MQLONG chlType;                         /* Channel type                    */
```

```
    MQLONG i;                               /* loop counter                    */
    MQLONG numberOfBags;                    /* number of bags in response bag  */
    MQCHAR chlName[MQ_OBJECT_NAME_LENGTH+1];/* name of chl extracted from bag  */
    MQCHAR OutputBuffer[100];               /* output data buffer              */
    OUTFILEHDL *outfp = NULL;               /* output file handle              */

    /***************************************************************************/
    /* Connect to the queue manager                                            */
    /***************************************************************************/
    if (argc &gt; 1)
       strncpy(qmName, argv[1], (size_t)MQ_Q_MGR_NAME_LENGTH);
    MQCONN(qmName, &hConn;, &compCode;, &connReason;);

    /***************************************************************************/
    /* Report the reason and stop if the connection failed.                    */
    /***************************************************************************/
    if (compCode == MQCC_FAILED)
    {
       CheckCallResult("Queue Manager connection", compCode, connReason);
       exit( (int)connReason);
    }

    /***************************************************************************/
    /* Open the output file                                                    */
    /***************************************************************************/
    if (argc &gt; 2)
    {
      OPENOUTFILE(outfp, argv[2]);
    }
    else
    {
      OPENOUTFILE(outfp, OUTFILE);
    }

    if(outfp == NULL)
    {
      printf("Could not open output file.\n");
      goto MOD_EXIT;
    }
    /***************************************************************************/
    /* Create an admin bag for the mqExecute call                              */
    /***************************************************************************/
    mqCreateBag(MQCBO_ADMIN_BAG, &adminBag;, &compCode;, &reason;);
    CheckCallResult("Create admin bag", compCode, reason);

    /***************************************************************************/
    /* Create a response bag for the mqExecute call                            */
    /***************************************************************************/
    mqCreateBag(MQCBO_ADMIN_BAG, &responseBag;, &compCode;, &reason;);
    CheckCallResult("Create response bag", compCode, reason);

    /***************************************************************************/
    /* Put the generic channel name into the admin bag                         */
    /***************************************************************************/
    mqAddString(adminBag, MQCACH_CHANNEL_NAME, MQBL_NULL_TERMINATED, "*",
              &compCode;, &reason;);
    CheckCallResult("Add channel name", compCode, reason);

    /***************************************************************************/
    /* Put the channel type into the admin bag                                 */
    /***************************************************************************/
    mqAddInteger(adminBag, MQIACH_CHANNEL_TYPE, MQCHT_ALL, &compCode;, &reason;);
    CheckCallResult("Add channel type", compCode, reason);

    /***************************************************************************/
    /* Add an inquiry for various attributes                                   */
    /***************************************************************************/
    mqAddInquiry(adminBag, MQIACH_CHANNEL_TYPE, &compCode;, &reason;);
    CheckCallResult("Add inquiry", compCode, reason);

    /***************************************************************************/
    /* Send the command to find all the channel names and channel types.       */
    /* The mqExecute call creates the PCF structure required, sends it to      */
    /* the command server, and receives the reply from the command server into */
    /* the response bag. The attributes are contained in system bags that are  */
    /* embedded in the response bag, one set of attributes per bag.            */
    /***************************************************************************/
    mqExecute(hConn,                   /* MQ connection handle                 */
              MQCMD_INQUIRE_CHANNEL,   /* Command to be executed               */
              MQHB_NONE,               /* No options bag                       */
              adminBag,                /* Handle to bag containing commands    */
              responseBag,             /* Handle to bag to receive the response*/
```

```
          MQHO_NONE,               /* Put msg on SYSTEM.ADMIN.COMMAND.QUEUE*/
          MQHO_NONE,               /* Create a dynamic q for the response  */
          &compCode;,               /* Completion code from the mqexecute   */
          &reason;);                /* Reason code from mqexecute call      */

/************************************************************************/
/* Check the command server is started. If not exit.                    */
/************************************************************************/
if (reason == MQRC_CMD_SERVER_NOT_AVAILABLE)
{
   printf("Please start the command server: <strmqcsv QMgrName="">\n");
   goto MOD_EXIT;
}

/************************************************************************/
/* Check the result from mqExecute call. If successful find the channel    */
/* types for all the channels. If failed find the error.                */
/************************************************************************/
if ( compCode == MQCC_OK )                      /* Successful mqExecute    */
{
  /**********************************************************************/
  /* Count the number of system bags embedded in the response bag from the */
  /* mqExecute call. The attributes for each channel are in separate bags. */
  /**********************************************************************/
  mqCountItems(responseBag, MQHA_BAG_HANDLE, &numberOfBags;,
               &compCode;, &reason;);
  CheckCallResult("Count number of bag handles", compCode, reason);

  for ( i=0; i<numberOfbags; i++)
  {
    /********************************************************************/
    /* Get the next system bag handle out of the mqExecute response bag.   */
    /* This bag contains the channel attributes                           */
    /********************************************************************/
    mqInquireBag(responseBag, MQHA_BAG_HANDLE, i, &cAttrsbag,
                 &compCode, &reason);
    CheckCallResult("Get the result bag handle", compCode, reason);

    /********************************************************************/
    /* Get the channel name out of the channel attributes bag             */
    /********************************************************************/
    mqInquireString(cAttrsBag, MQCACH_CHANNEL_NAME, 0, MQ_OBJECT_NAME_LENGTH,
                    chlName, &chlNameLength, NULL, &compCode, &reason);
    CheckCallResult("Get channel name", compCode, reason);

    /********************************************************************/
    /* Get the channel type out of the channel attributes bag             */
    /********************************************************************/

 mqInquireInteger(cAttrsBag, MQIACH_CHANNEL_TYPE, MQIND_NONE, &chlType,
                    &compCode, &reason);
    CheckCallResult("Get type", compCode, reason);

    /********************************************************************/
    /* Use mqTrim to prepare the channel name for printing.               */
    /* Print the result.                                                  */
    /********************************************************************/
    mqTrim(MQ_CHANNEL_NAME_LENGTH, chlName, chlName, &compCode, &reason);
    sprintf(OutputBuffer, "%-20s%-9s", chlName, ChlType2String(chlType));
    WRITEOUTFILE(outfp,OutputBuffer,29)
  }
}

else                                               /* Failed mqExecute     */
{
  printf("Call to get channel attributes failed: Cc = %ld : Rc = %ld\n",
               compCode, reason);
  /**********************************************************************/
  /* If the command fails get the system bag handle out of the mqexecute   */
  /* response bag.This bag contains the reason from the command server     */
  /* why the command failed.                                              */
  /**********************************************************************/
  if (reason == MQRCCF_COMMAND_FAILED)
  {
    mqInquireBag(responseBag, MQHA_BAG_HANDLE, 0, &errorBag,
                 &compCode, &reason);
    CheckCallResult("Get the result bag handle", compCode, reason);

    /********************************************************************/
    /* Get the completion code and reason code, returned by the command    */
    /* server, from the embedded error bag.                               */
    /********************************************************************/
```

```
        mqInquireInteger(errorBag, MQIASY_COMP_CODE, MQIND_NONE, &mqExecuteCC,
                         &compCode, &reason );
        CheckCallResult("Get the completion code from the result bag",
                        compCode, reason);
        mqInquireInteger(errorBag, MQIASY_REASON, MQIND_NONE, &mqExecuteRC,
                          &compCode, &reason);
        CheckCallResult("Get the reason code from the result bag",
                        compCode, reason);
        printf("Error returned by the command server: Cc = %ld : Rc = %ld\n",
                 mqExecuteCC, mqExecuteRC);
      }
    }

MOD_EXIT:
   /***************************************************************************/
   /* Delete the admin bag if successfully created.                           */
   /***************************************************************************/
   if (adminBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&adminBag, &compCode, &reason);
      CheckCallResult("Delete the admin bag", compCode, reason);
   }

   /***************************************************************************/
   /* Delete the response bag if successfully created.                        */
   /***************************************************************************/
   if (responseBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&responseBag, &compCode, &reason);
      CheckCallResult("Delete the response bag", compCode, reason);
   }

   /***************************************************************************/
   /* Disconnect from the queue manager if not already connected              */
   /***************************************************************************/
   if (connReason != MQRC_ALREADY_CONNECTED)
   {
      MQDISC(&hConn, &compCode, &reason);
       CheckCallResult("Disconnect from Queue Manager", compCode, reason);
   }

   /***************************************************************************/
   /* Close the output file if open                                           */
   /***************************************************************************/
   if(outfp != NULL)
      CLOSEOUTFILE(outfp);

   return 0;
}


/******************************************************************************/
/*                                                                            */
/* Function: CheckCallResult                                                  */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Input Parameters:  Description of call                                     */
/*                    Completion code                                         */
/*                    Reason code                                             */
/*                                                                            */
/* Output Parameters: None                                                    */
/*                                                                            */
/* Logic: Display the description of the call, the completion code and the    */
/*        reason code if the completion code is not successful                */
/*                                                                            */
/******************************************************************************/
void  CheckCallResult(char *callText, MQLONG cc, MQLONG rc)
{
   if (cc != MQCC_OK)
         printf("%s failed: Completion Code = %ld : Reason = %ld\n", callText,
                cc, rc);
}
```

Sample C program for inquiring about queues and printing information (amqsailq.c)

The sample C program `amqsailq.c` inquires the current depth of the local queues using the MQAI.

```
/******************************************************************************/
/*                                                                            */
/* Program name: AMQSAILQ.C                                                   */
/*                                                                            */
/* Description:  Sample C program to inquire the current depth of the local   */
/*               queues using the WebSphere MQ Administration Interface (MQAI)*/
/*                                                                            */
/* Statement:    Licensed Materials - Property of IBM                         */
/*                                                                            */
/*               84H2000, 5765-B73                                            */
/*               84H2001, 5639-B42                                            */
/*               84H2002, 5765-B74                                            */
/*               84H2003, 5765-B75                                            */
/*               84H2004, 5639-B43                                            */
/*                                                                            */
/*               (C) Copyright IBM Corp. 1999, 2025.                          */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* Function:                                                                  */
/*    AMQSAILQ is a sample C program that demonstrates how to inquire         */
/*    attributes of the local queue manager using the MQAI interface. In      */
/*    particular, it inquires the current depths of all the local queues.     */
/*                                                                            */
/*     - A PCF command is built by placing items into an MQAI administration  */
/*       bag.                                                                 */
/*       These are:-                                                          */
/*            - The generic queue name "*"                                    */
/*            - The type of queue required. In this sample we want to         */
/*              inquire local queues.                                         */
/*            - The attribute to be inquired. In this sample we want the      */
/*              current depths.                                               */
/*                                                                            */
/*     - The mqExecute call is executed with the command MQCMD_INQUIRE_Q.     */
/*       The call generates the correct PCF structure.                        */
/*       The default options to the call are used so that the command is sent */
/*       to the SYSTEM.ADMIN.COMMAND.QUEUE.                                   */
/*       The reply from the command server is placed on a temporary dynamic   */
/*       queue.                                                               */
/*       The reply from the MQCMD_INQUIRE_Q command is read from the          */
/*       temporary queue and formatted into the response bag.                 */
/*                                                                            */
/*     - The completion code from the mqExecute call is checked and if there  */
/*       is a failure from the command server, then the code returned by      */
/*       command server is retrieved from the system bag that has been        */
/*       embedded in the response bag to the mqExecute call.                  */
/*                                                                            */
/*     - If the call is successful, the depth of each local queue is placed   */
/*       in system bags embedded in the response bag of the mqExecute call.   */
/*       The name and depth of each queue is obtained from each of the bags   */
/*       and the result displayed on the screen.                              */
/*                                                                            */
/* Note: The command server must be running.                                  */
/*                                                                            */
/******************************************************************************/
/*                                                                            */
/* AMQSAILQ has 1 parameter - the queue manager name (optional)               */
/*                                                                            */
/******************************************************************************/

/******************************************************************************/
/* Includes                                                                   */
/******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#include <cmqc.h>                          /* MQI                             */
#include <cmqcfc.h>                        /* PCF                             */
#include <cmqbc.h>                         /* MQAI                            */

/******************************************************************************/
```

```
/* Function prototypes                                                      */
/****************************************************************************/
void CheckCallResult(MQCHAR *, MQLONG , MQLONG);

/****************************************************************************/
/* Function: main                                                           */
/****************************************************************************/
int main(int argc, char *argv[])
{
   /*************************************************************************/
   /* MQAI variables                                                        */
   /*************************************************************************/
   MQHCONN hConn;                      /* handle to WebSphere MQ connection   */
   MQCHAR qmName[MQ_Q_MGR_NAME_LENGTH+1]=""; /* default QMgr name             */
   MQLONG reason;                      /* reason code                        */
   MQLONG connReason;                  /* MQCONN reason code                 */
   MQLONG compCode;                    /* completion code                    */
   MQHBAG adminBag = MQHB_UNUSABLE_HBAG;   /* admin bag for mqExecute        */
   MQHBAG responseBag = MQHB_UNUSABLE_HBAG;/* response bag for mqExecute     */
   MQHBAG qAttrsBag;                   /* bag containing q attributes        */
   MQHBAG errorBag;                    /* bag containing cmd server error */
   MQLONG mqExecuteCC;                 /* mqExecute completion code          */
   MQLONG mqExecuteRC;                 /* mqExecute reason code              */
   MQLONG qNameLength;                 /* Actual length of q name            */
   MQLONG qDepth;                      /* depth of queue                     */
   MQLONG i;                           /* loop counter                       */
   MQLONG numberOfBags;                /* number of bags in response bag  */
   MQCHAR qName[MQ_Q_NAME_LENGTH+1];       /* name of queue extracted from bag*/


   printf("Display current depths of local queues\n\n");

   /*************************************************************************/
   /* Connect to the queue manager                                          */
   /*************************************************************************/
   if (argc > 1)
      strncpy(qmName, argv[1], (size_t)MQ_Q_MGR_NAME_LENGTH);
   MQCONN(qmName, &hConn, &compCode, &connReason);

   /*************************************************************************/
   /* Report the reason and stop if the connection failed.                  */
   /*************************************************************************/
   if (compCode == MQCC_FAILED)
   {
      CheckCallResult("Queue Manager connection", compCode, connReason
);
      exit( (int)connReason);
   }

   /*************************************************************************/
   /* Create an admin bag for the mqExecute call                            */
   /*************************************************************************/
   mqCreateBag(MQCBO_ADMIN_BAG, &adminBag, &compCode, &reason);
   CheckCallResult("Create admin bag", compCode, reason);
   /*************************************************************************/
   /* Create a response bag for the mqExecute call                          */
   /*************************************************************************/
   mqCreateBag(MQCBO_ADMIN_BAG, &responseBag, &compCode, &reason);
   CheckCallResult("Create response bag", compCode, reason);

   /*************************************************************************/
   /* Put the generic queue name into the admin bag                         */
   /*************************************************************************/
   mqAddString(adminBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED, "*",
               &compCode, &reason);
   CheckCallResult("Add q name", compCode, reason);

   /*************************************************************************/
   /* Put the local queue type into the admin bag                           */
   /*************************************************************************/
   mqAddInteger(adminBag, MQIA_Q_TYPE, MQQT_LOCAL, &compCode, &reason);
   CheckCallResult("Add q type", compCode, reason);

   /*************************************************************************/
   /* Add an inquiry for current queue depths                               */
   /*************************************************************************/
   mqAddInquiry(adminBag, MQIA_CURRENT_Q_DEPTH, &compCode, &reason);
   CheckCallResult("Add inquiry", compCode, reason);

   /*************************************************************************/
   /* Send the command to find all the local queue names and queue depths.  */
   /* The mqExecute call creates the PCF structure required, sends it to    */
/* the command server, and receives the reply from the command server into */
/* the response bag. The attributes are contained in system bags that are  */
/* embedded in the response bag, one set of attributes per bag.            */
/***************************************************************************/
mqExecute(hConn,                   /* WebSphere MQ connection handle           */
          MQCMD_INQUIRE_Q,         /* Command to be executed               */
          MQHB_NONE,               /* No options bag                       */
          adminBag,                /* Handle to bag containing commands    */
          responseBag,             /* Handle to bag to receive the response*/
          MQHO_NONE,               /* Put msg on SYSTEM.ADMIN.COMMAND.QUEUE*/
          MQHO_NONE,               /* Create a dynamic q for the response  */
          &compCode,               /* Completion code from the mqExecute   */
          &reason);                /* Reason code from mqExecute call      */


/***************************************************************************/
/* Check the command server is started. If not exit.                       */
/***************************************************************************/
if (reason == MQRC_CMD_SERVER_NOT_AVAILABLE)
{
   printf("Please start the command server: <strmqcsv QMgrName>\n");
   MQDISC(&hConn, &compCode, &reason);
   CheckCallResult("Disconnect from Queue Manager", compCode, reason);
   exit(98);
}

/***************************************************************************/
/* Check the result from mqExecute call. If successful find the current    */
/* depths of all the local queues. If failed find the error.               */
/***************************************************************************/
if ( compCode == MQCC_OK )                      /* Successful mqExecute    */
{
  /*************************************************************************/
  /* Count the number of system bags embedded in the response bag from the */
  /* mqExecute call. The attributes for each queue are in a separate bag.  */
  /*************************************************************************/
  mqCountItems(responseBag, MQHA_BAG_HANDLE, &numberOfBags, &compCode,
               &reason);
  CheckCallResult("Count number of bag handles", compCode, reason);

  for ( i=0; i<numberOfBags; i++)
  {
    /***********************************************************************/
    /* Get the next system bag handle out of the mqExecute response bag.   */
    /* This bag contains the queue attributes                              */
    /***********************************************************************/
    mqInquireBag(responseBag, MQHA_BAG_HANDLE, i, &qAttrsBag, &compCode,
                 &reason);
    CheckCallResult("Get the result bag handle", compCode, reason);

    /***********************************************************************/
    /* Get the queue name out of the queue attributes bag                  */
    /***********************************************************************/
    mqInquireString(qAttrsBag, MQCA_Q_NAME, 0, MQ_Q_NAME_LENGTH, qName,
                    &qNameLength, NULL, &compCode, &reason);
    CheckCallResult("Get queue name", compCode, reason);

    /***********************************************************************/
    /* Get the depth out of the queue attributes bag                       */
    /***********************************************************************/
    mqInquireInteger(qAttrsBag, MQIA_CURRENT_Q_DEPTH, MQIND_NONE, &qDepth,
                     &compCode, &reason);
    CheckCallResult("Get depth", compCode, reason);

    /***********************************************************************/
    /* Use mqTrim to prepare the queue name for printing.                  */
    /* Print the result.                                                   */
    /***********************************************************************/
    mqTrim(MQ_Q_NAME_LENGTH, qName, qName, &compCode, &reason)
    printf("%4d  %-48s\n", qDepth, qName);
  }
}

else                                               /* Failed mqExecute     */
{
  printf("Call to get queue attributes failed: Completion Code = %d :
          Reason = %d\n", compCode, reason);

  /*************************************************************************/
  /* If the command fails get the system bag handle out of the mqExecute   */
  /* response bag. This bag contains the reason from the command server    */
  /* why the command failed.                                               */
```

```
    /***********************************************************************/
    if (reason == MQRCCF_COMMAND_FAILED)
    {
      mqInquireBag(responseBag, MQHA_BAG_HANDLE, 0, &errorBag, &compCode,
                   &reason);
      CheckCallResult("Get the result bag handle", compCode, reason);

     /*********************************************************************/
     /* Get the completion code and reason code, returned by the command   */
     /* server, from the embedded error bag.                               */
     /*********************************************************************/
     mqInquireInteger(errorBag, MQIASY_COMP_CODE, MQIND_NONE, &mqExecuteCC,
                      &compCode, &reason );
     CheckCallResult("Get the completion code from the result bag",
                      compCode, reason);
     mqInquireInteger(errorBag, MQIASY_REASON, MQIND_NONE, &mqExecuteRC,
                       &compCode, &reason);
     CheckCallResult("Get the reason code from the result bag",
                      compCode, reason);
     printf("Error returned by the command server: Completion Code = %d :
             Reason = %d\n", mqExecuteCC, mqExecuteRC);
   }
 }

 /***************************************************************************/
 /* Delete the admin bag if successfully created.                           */
 /***************************************************************************/
 if (adminBag != MQHB_UNUSABLE_HBAG)
 {
    mqDeleteBag(&adminBag, &compCode, &reason);
    CheckCallResult("Delete the admin bag", compCode, reason);
 }

 /***************************************************************************/
 /* Delete the response bag if successfully created.                        */
 /***************************************************************************/
 if (responseBag != MQHB_UNUSABLE_HBAG)
 {
    mqDeleteBag(&responseBag, &compCode, &reason);
    CheckCallResult("Delete the response bag", compCode, reason);
 }

 /***************************************************************************/
 /* Disconnect from the queue manager if not already connected              */
 /***************************************************************************/
 if (connReason != MQRC_ALREADY_CONNECTED)
 {
    MQDISC(&hConn, &compCode, &reason);
     CheckCallResult("Disconnect from queue manager", compCode, reason);
 }
 return 0;
}

*******************************************************************************/
*                                                                             */
* Function: CheckCallResult                                                   */
*                                                                             */
*******************************************************************************/
*                                                                             */
* Input Parameters:  Description of call                                     */
*                    Completion code                                          */
*                    Reason code                                              */
*                                                                             */
* Output Parameters: None                                                     */
*                                                                             */
* Logic: Display the description of the call, the completion code and the    */
*        reason code if the completion code is not successful                 */
*                                                                             */
*******************************************************************************/
void  CheckCallResult(char *callText, MQLONG cc, MQLONG rc)
{
 if (cc != MQCC_OK)
       printf("%s failed: Completion Code = %d : Reason = %d\n",
               callText, cc, rc);
}
```

Hints and tips for configuring IBM WebSphere MQ

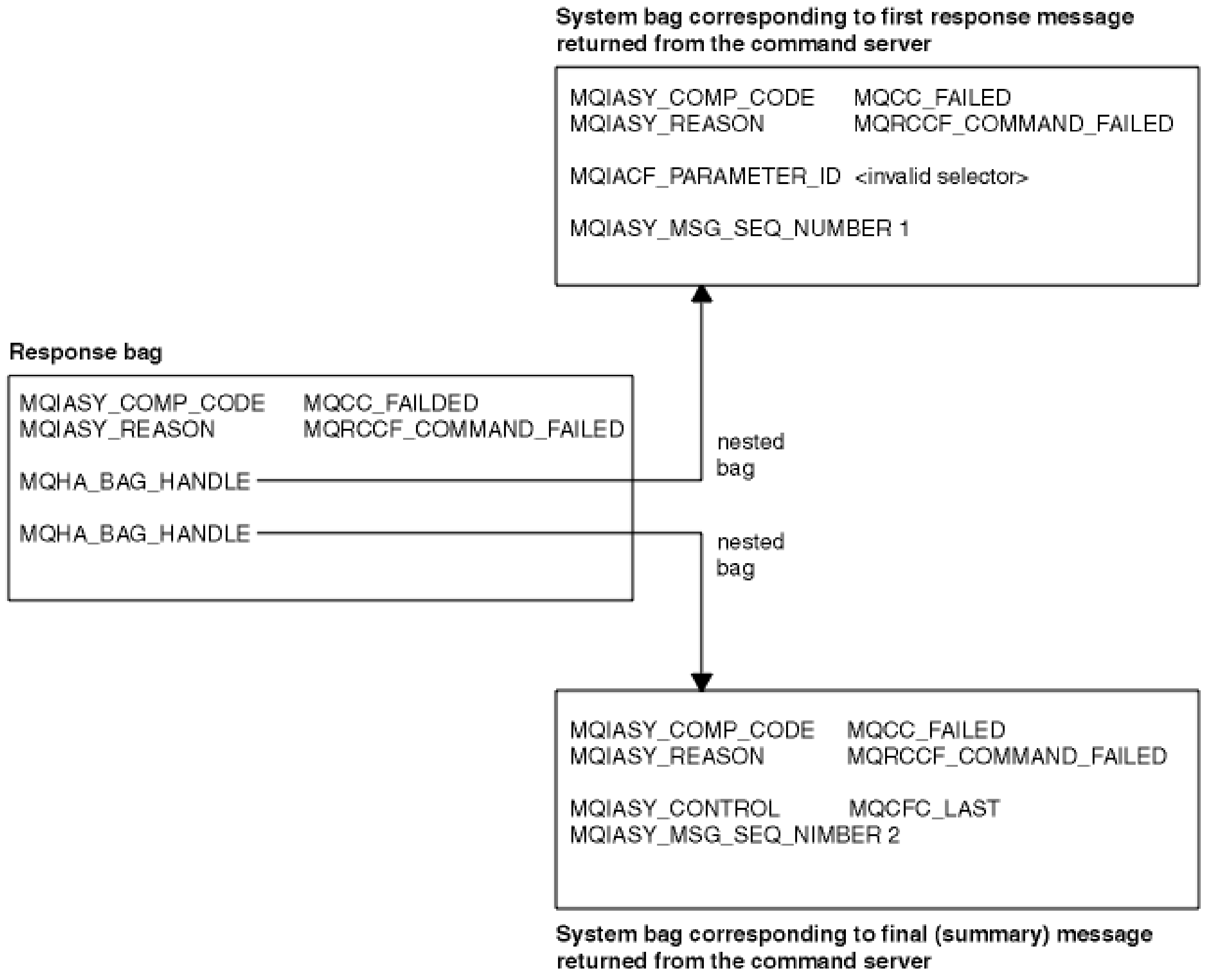
Programming hints and tips when using the MQAI.

The MQAI uses PCF messages to send administration commands to the command server rather than dealing directly with the command server itself. Here are some tips for configuring WebSphere MQ using the MQAI:

- Character strings in WebSphere MQ are blank padded to a fixed length. Using C, null-terminated strings can normally be supplied as input parameters to WebSphere MQ programming interfaces.
- To clear the value of a string attribute, set it to a single blank rather than an empty string.
- Consider in advance the attributes that you want to change and inquire on just those attributes.
- Certain attributes cannot be changed, for example a queue name or a channel type. Ensure that you attempt to change only those attributes that can be modified. Refer to the list of required and optional parameters for the specific PCF change object. See Definitions of the Programmable Command Formats.
- If an MQAI call fails, some detail of the failure is returned to the response bag. Further detail can then be found in a nested bag that can be accessed by the selector MQHA_BAG_HANDLE. For example, if an mqExecute call fails with a reason code of MQRCCF_COMMAND_FAILED, this information is returned in the response bag. A possible reason for this reason code is that a selector specified was not valid for the type of command message and this detail of information is found in a nested bag that can be accessed by a bag handle.

  For more information on MQExecute, see "Sending administration commands to the command server using the mqExecute call" on page 54

  The following diagram shows this scenario:


System bag corresponding to first response message returned from the command server
MQIASY_COMP_CODE MQCC_FAILED
MQIASY_REASON MQRCCF_COMMAND_FAILED
MQIACF_PARAMETER_ID <invalid selector>
MQIASY_MSG_SEQ_NUMBER 1
Response bag
MQIASY_COMP_CODE MQCC_FAILDED
MQIASY_REASON MQRCCF_COMMAND_FAILED
MQHA_BAG_HANDLE
MQHA_BAG_HANDLE
nested bag
nested bag
MQIASY_COMP_CODE MQCC_FAILED
MQIASY_REASON MQRCCF_COMMAND_FAILED
MQIASY_CONTROL MQCFC_LAST
MQIASY_MSG_SEQ_NIMBER 2
System bag corresponding to final (summary) message returned from the command server

Advanced MQAI topics

Information on indexing, data conversion and use of message descriptor

- Indexing

  Indexes are used when replacing or removing existing data items from a bag to preserve insertion order. Full details on indexing can be found in "Indexing in the MQAI" on page 42.
- Data conversion

  The strings contained in an MQAI data bag can be in a variety of coded character sets and these can be converted using the mqSetInteger call. Full details on data conversion can be found in "Data conversion in the MQAI" on page 43.
- Use of the message descriptor

  MQAI generates a message descriptor which is set to an initial value when the data bag is created. Full details of the use of the message descriptor can be found in "Use of the message descriptor in the MQAI" on page 45.

Indexing in the MQAI

Indexes are used when replacing or removing existing data items from a bag. There are three types of indexing, which allows data items to be retrieved easily.

Each selector and value within a data item in a bag have three associated index numbers:

- The index relative to other items that have the same selector.

- The index relative to the category of selector (user or system) to which the item belongs.
- The index relative to all the data items in the bag (user and system).

This allows indexing by user selectors, system selectors, or both as shown in Figure 1 on page 43.

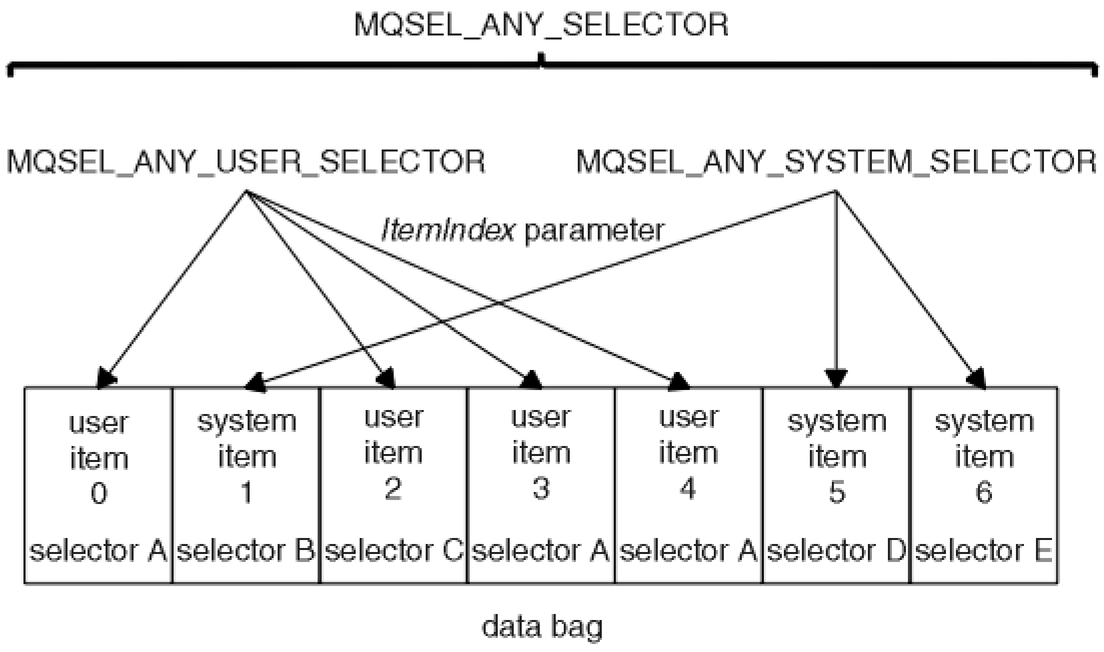


Figure 1. Indexing

In Figure Figure 1 on page 43, user item 3 (selector A) can be referred to by the following index pairs:

Selector	***ItemIndex***
selector A	1
MQSEL_ANY_USER_SELECTOR	2
MQSEL_ANY_SELECTOR	3

The index is zero-based like an array in C; if there are 'n' occurrences, the index ranges from zero through 'n-1', with no gaps.

Indexes are used when replacing or removing existing data items from a bag. When used in this way, the insertion order is preserved, but indexes of other data items can be affected. For examples of this, see Changing information within a bag and Deleting data items.

The three types of indexing allow easy retrieval of data items. For example, if there are three instances of a particular selector in a bag, the mqCountItems call can count the number of instances of that selector, and the mqInquire* calls can specify both the selector and the index to inquire those values only. This is useful for attributes that can have a list of values such as some of the exits on channels.

Data conversion in the MQAI

The strings contained in an MQAI data bag can be in a variety of coded character sets. These strings can be converted using the mqSetInteger call.

Like PCF messages, the strings contained in an MQAI data bag can be in a variety of coded character sets. Usually, all of the strings in a PCF message are in the same coded character set; that is, the same set as the queue manager.

Each string item in a data bag contains two values; the string itself and the CCSID. The string that is added to the bag is obtained from the *Buffer* parameter of the mqAddString or mqSetString call. The CCSID is obtained from the system item containing a selector of MQIASY_CODED_CHAR_SET_ID. This is known as the *bag CCSID* and can be changed using the mqSetInteger call.

When you inquire the value of a string contained in a data bag, the CCSID is an output parameter from the call.

Table 2 on page 44 shows the rules applied when converting data bags into messages and vice versa:

Table 2. CCSID processing

MQAI call	CCSID	Input to call	Output to call
mqBagToBuffer	Bag CCSID (1)	Ignored	Unchanged
mqBagToBuffer	String CCSIDs in bag	Used	Unchanged
mqBagToBuffer	String CCSIDs in buffer	Not applicable	Copied from string CCSIDs in bag
mqBufferToBag	Bag CCSID (1)	Ignored	Unchanged
mqBufferToBag	String CCSIDs in buffer	Used	Unchanged
mqBufferToBag	String CCSIDs in bag	Not applicable	Copied from string CCSIDs in buffer
mqPutBag	MQMD CCSID	Used	Unchanged (2)
mqPutBag	Bag CCSID (1)	Ignored	Unchanged
mqPutBag	String CCSIDs in bag	Used	Unchanged
mqPutBag	String CCSIDs in message sent	Not applicable	Copied from string CCSIDs in bag
mqGetBag	MQMD CCSID	Used for data conversion of message	Set to CCSID of data returned (3)
mqGetBag	Bag CCSID (1)	Ignored	Unchanged
mqGetBag	String CCSIDs in message	Used	Unchanged
mqGetBag	String CCSIDs in bag	Not applicable	Copied from string CCSIDs in message
mqExecute	Request-bag CCSID	Used for MQMD of request message (4)	Unchanged
mqExecute	Reply-bag CCSID	Used for data conversion of reply message (4)	Set to CCSID of data returned (3)
mqExecute	String CCSIDs in request bag	Used for request message	Unchanged
mqExecute	String CCSIDs in reply bag	Not applicable	Copied from string CCSIDs in reply message

Notes:

1. Bag CCSID is the system item with selector MQIASY_CODED_CHAR_SET_ID.
2. MQCCSI_Q_MGR is changed to the actual queue manager CCSID.
3. If data conversion is requested, the CCSID of data returned is the same as the output value. If data conversion is not requested, the CCSID of data returned is the same as the message value. Note that no message is returned if data conversion is requested but fails.
4. If the CCSID is MQCCSI_DEFAULT, the queue manager's CCSID is used.

Use of the message descriptor in the MQAI

The message descriptor that the MQAI generates is set to an initial value when the data bag is created.

The PCF command type is obtained from the system item with selector MQIASY_TYPE. When you create your data bag, the initial value of this item is set depending on the type of bag you create:

Table 3. PCF command type

Type of bag	Initial value of MQIASY_TYPE item
MQCBO_ADMIN_BAG	MQCFT_COMMAND
MQCBO_COMMAND_BAG	MQCFT_COMMAND
MQCBO_*	MQCFT_USER

When the MQAI generates a message descriptor, the values used in the *Format* and *MsgType* parameters depend on the value of the system item with selector MQIASY_TYPE as shown in Table 3 on page 45.

Table 4. Format and MsgType parameters of the MQMD

PCF command type	Format	MsgType
MQCFT_COMMAND	MQFMT_ADMIN	MQMT_REQUEST
MQCFT_REPORT	MQFMT_ADMIN	MQMT_REPORT
MQCFT_RESPONSE	MQFMT_ADMIN	MQMT_REPLY
MQCFT_TRACE_ROUTE	MQFMT_ADMIN	MQMT_DATAGRAM
MQCFT_EVENT	MQFMT_EVENT	MQMT_DATAGRAM
MQCFT_*	MQFMT_PCF	MQMT_DATAGRAM

Table 4 on page 45 shows that if you create an administration bag or a command bag, the *Format* of the message descriptor is MQFMT_ADMIN and the *MsgType* is MQMT_REQUEST. This is suitable for a PCF request message sent to the command server when a response is expected back.

Other parameters in the message descriptor take the values shown in Table 5 on page 45.

Table 5. Message descriptor values

Parameter	Value
StrucId	MQMD_STRUC_ID
Version	MQMD_VERSION_1
Report	MQRO_NONE
MsgType	see Table 4 on page 45
Expiry	30 seconds (note “1” on page 46)
Feedback	MQFB_NONE
Encoding	MQENC_NATIVE
CodedCharSetId	depends on the bag CCSID (note “2” on page 46)
Format	see Table 4 on page 45
Priority	MQPRI_PRIORITY_AS_Q_DEF
Persistence	MQPER_NOT_PERSISTENT
MsgId	MQMI_NONE

Table 5. Message descriptor values (continued)

Parameter	**Value**
CorrelId	MQCI_NONE
BackoutCount	0
ReplyToQ	see note "3" on page 46
ReplyToQMgr	blank

Notes:

1. This value can be overridden on the **mqExecute** call by using the **OptionsBag** parameter. For information about this, see **mqExecute**.
2. See "Data conversion in the MQAI" on page 43.
3. Name of the user-specified reply queue or MQAI-generated temporary dynamic queue for messages of type **MQMT_REQUEST**. Blank otherwise.

Data bags

A data bag is a means of handling properties or parameters of objects using the MQAI.

Data Bags

- The data bag contains zero or more *data items*. These data items are ordered within the bag as they are placed into the bag. This is called the *insertion order*. Each data item contains a *selector* that identifies the data item and a *value* of that data item that can be either an integer, a 64-bit integer, an integer filter, a string, a string filter, a byte string, a byte string filter, or a handle of another bag. Data items are described in details in "Data item" on page 49

  There are two types of selector; *user selectors* and *system selectors*. These are described in MQAI Selectors. The selectors are usually unique, but it is possible to have multiple values for the same selector. In this case, an *index* identifies the particular occurrence of selector that is required. Indexes are described in "Indexing in the MQAI" on page 42.

  A hierarchy of the these concepts is shown in Figure 1.

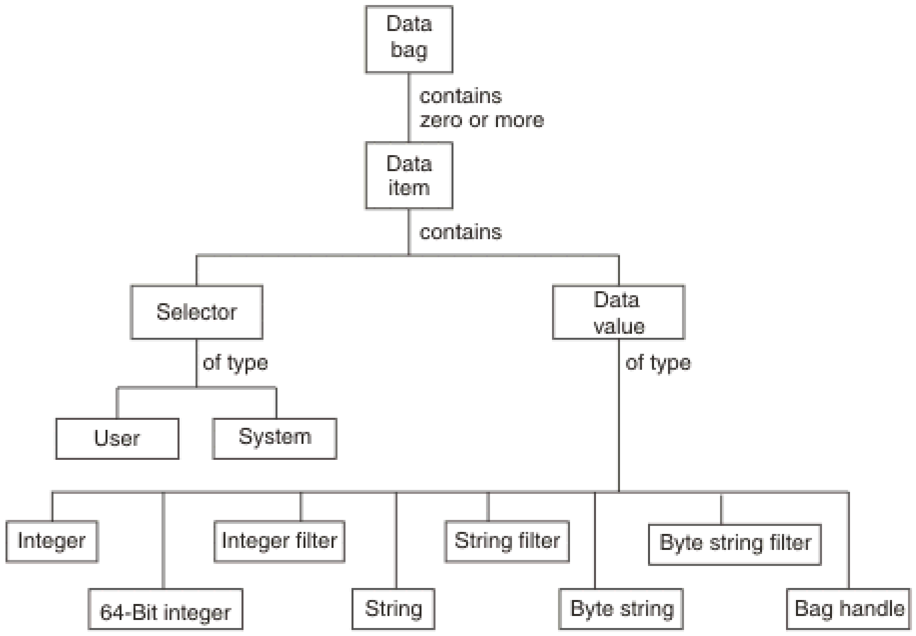


Figure 2. Hierarchy of MQAI concepts

The hierarchy has been explained in a previous paragraph.

Types of data bag

You can choose the type of data bag that you want to create depending on the task that you wish to perform:

user bag
A simple bag used for user data.

administration bag
A bag created for data used to administer WebSphere MQ objects by sending administration messages to a command server. The administration bag automatically implies certain options as described in "Creating and deleting data bags" on page 48.

command bag
A bag also created for commands for administering WebSphere MQ objects. However, unlike the administration bag, the command bag does not automatically imply certain options although these options are available. For more information about options, see "Creating and deleting data bags" on page 48.

group bag
A bag used to hold a set of grouped data items. Group bags cannot be used for administering WebSphere MQ objects.

In addition, the **system bag** is created by the MQAI when a reply message is returned from the command server and placed into a user's output bag. A system bag cannot be modified by the user.

Using Data Bags The different ways of using data bags are listed in this topic:

Using Data Bags

The different ways of using data bags are shown in the following list:

- You can create and delete data bags "Creating and deleting data bags" on page 48.
- You can send data between applications using data bags "Putting and receiving data bags" on page 48.
- You can add data items to data bags "Adding data items to bags" on page 49.

- You can add an inquiry command within a data bag “Adding an inquiry command to a bag” on page 50.
- You can inquire within data bags “Inquiring within data bags” on page 51.
- You can count data items within a data bag “Counting data items” on page 53.
- You can change information within a data bag “Changing information within a bag” on page 51.
- You can clear a data bag “Clearing a bag using the mqClearBag call” on page 52.
- You can truncate a data bag “Truncating a bag using the mqTruncateBag call” on page 52.
- You can convert bags and buffers “Converting bags and buffers” on page 53.

Creating and deleting data bags

Creating data bags

To use the MQAI, you first create a data bag using the mqCreateBag call. As input to this call, you supply one or more options to control the creation of the bag.

The `Options` parameter of the MQCreateBag call lets you choose whether to create a user bag, a command bag, a group bag, or an administration bag.

To create a user bag, a command bag, or a group bag, you can choose one or more further options to:

- Use the list form when there are two or more adjacent occurrences of the same selector in a bag.
- Reorder the data items as they are added to a PCF message to ensure that the parameters are in their correct order. For more information on data items, see “Data item” on page 49.
- Check the values of user selectors for items that you add to the bag.

Administration bags automatically imply these options.

A data bag is identified by its handle. The bag handle is returned from mqCreateBag and must be supplied on all other calls that use the data bag.

For a full description of the mqCreateBag call, see mqCreateBag.

Deleting data bags

Any data bag that is created by the user must also be deleted using the mqDeleteBag call. For example, if a bag is created in the user code, it must also be deleted in the user code.

System bags are created and deleted automatically by the MQAI. For more information about this, see “Sending administration commands to the command server using the mqExecute call” on page 54. User code cannot delete a system bag.

For a full description of the mqDeleteBag call, see mqDeleteBag.

Putting and receiving data bags

Data can also be sent between applications by putting and getting data bags using the mqPutBag and mqGetBag calls. This lets the MQAI handle the buffer rather than the application. The mqPutBag call converts the contents of the specified bag into a PCF message and sends the message to the specified queue and the mqGetBag call removes the message from the specified queue and converts it back into a data bag. Therefore, the mqPutBag call is the equivalent of the mqBagToBuffer call followed by MQPUT, and the mqGetBag is the equivalent of the MQGET call followed by mqBufferToBag.

For more information on sending and receiving PCF messages in a specific queue, see “Sending and receiving PCF messages in a specified queue” on page 12

Note: If you choose to use the mqGetBag call, the PCF details within the message must be correct; if they are not, an appropriate error results and the PCF message is not returned.

Data item

Data items are used to populate Data bags when they are created. These data items can be user or system items.

These user items contain user data such as attributes of objects that are being administered. System items should be used for more control over the messages generated: for example, the generation of message headers. For more information about system items, see "System items" on page 49.

Types of Data Items

When you have created a data bag, you can populate it with integer or character-string items. You can inquire about all three types of item.

The data item can either be integer or character-string items. Here are the types of data item available within the MQAI:

- Integer
- 64-bit integer
- Integer filter
- Character-string
- String filter
- Byte string
- Byte string filter
- Bag handle

Using Data Items

These are the following ways of using data items:

- "Counting data items" on page 53.
- "Deleting data items" on page 53.
- "Adding data items to bags" on page 49.
- "Filtering and querying data items" on page 50.

System items

System items can be used for:

- The generation of PCF headers. System items can control the PCF command identifier, control options, message sequence number, and command type.
- Data conversion. System items handle the character-set identifier for the character-string items in the bag.

Like all data items, system items consist of a selector and a value. For information about these selectors and what they are for, see MQAI Selectors.

System items are unique. One or more system items can be identified by a system selector. There is only one occurrence of each system selector.

Most system items can be modified (see "Changing information within a bag" on page 51), but the bag-creation options cannot be changed by the user. You cannot delete system items. (See "Deleting data items" on page 53.)

Adding data items to bags

When a data bag is created, you can populate it with data items. These data items can be user or system items. For more information about data items, see "Data item" on page 49.

The MQAI lets you add integer items, 64-bit integer items, integer filter items, character-string items, string filter, byte string items, and byte string filter items to bags and this is shown in Figure 3 on page

50. The items are identified by a selector. Usually one selector identifies one item only, but this is not always the case. If a data item with the specified selector is already present in the bag, an additional instance of that selector is added to the end of the bag.

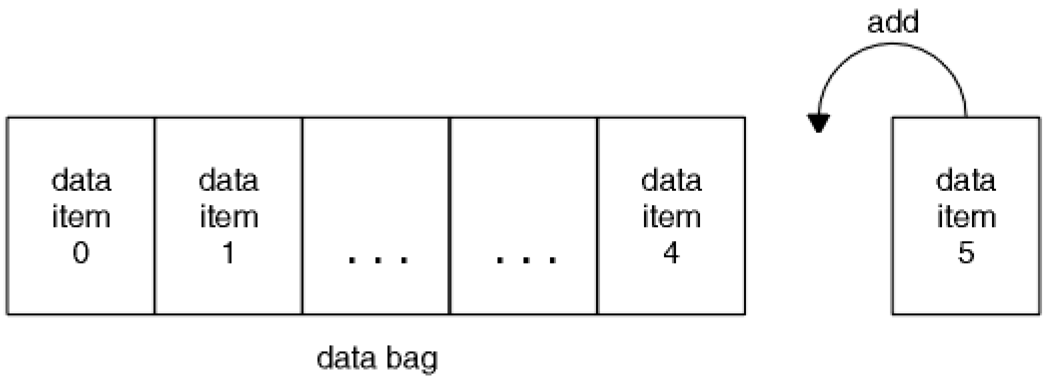


Figure 3. Adding data items

Add data items to a bag using the mqAdd* calls:

- To add integer items, use the mqAddInteger call as described in mqAddInteger
- To add 64-bit integer items, use the mqAddInteger64 call as described in mqAddInteger64
- To add integer filter items, use the mqAddIntegerFilter call as described in mqAddIntegerFilter
- To add character-string items, use the mqAddString call as described in mqAddString
- To add string filter items, use the mqAddStringFilter call as described in mqAddStringFilter
- To add byte string items, use the mqAddByteString call as described in mqAddByteString
- To add byte string filter items, use the mqAddByteStringFilter call as described in mqAddByteStringFilter

For more information on adding data items to a bag, see "System items" on page 49.

Adding an inquiry command to a bag

The mqAddInquiry call is used to add an inquiry command to a bag. The call is specifically for administration purposes, so it can be used with administration bags only. It lets you specify the selectors of attributes on which you want to inquire from WebSphere MQ.

For a full description of the mqAddInquiry call, see mqAddInquiry.

Filtering and querying data items

When using the MQAI to inquire about the attributes of WebSphere MQ objects, you can control the data that is returned to your program in two ways.

- You can ***filter*** the data that is returned using the mqAddInteger and mqAddString calls. This approach lets you specify a *Selector* and *ItemValue* pair, for example:

  ```
  mqAddInteger(inputbag, MQIA_Q_TYPE, MQQT_LOCAL)
  ```

  This example specifies that the queue type (*Selector*) must be local (*ItemValue*) and this specification must match the attributes of the object (in this case, a queue) about which you are inquiring.

  Other attributes that can be filtered correspond to the PCF Inquire* commands that can be found in "Introduction to Programmable Command Formats" on page 9. For example, to inquire about the attributes of a channel, see the Inquire Channel command in this product documentation. The "Required parameters" and "Optional parameters" of the Inquire Channel command identify the selectors that you can use for filtering.

- You can ***query*** particular attributes of an object using the mqAddInquiry call. This specifies the selector in which you are interested. If you do not specify the selector, all attributes of the object are returned.

Here is an example of filtering and querying the attributes of a queue:

```
/* Request information about all queues */
mqAddString(adminbag, MQCA_Q_NAME, "*")

/* Filter attributes so that local queues only are returned */
mqAddInteger(adminbag, MQIA_Q_TYPE, MQQT_LOCAL)

/* Query the names and current depths of the local queues */
mqAddInquiry(adminbag, MQCA_Q_NAME)
mqAddInquiry(adminbag, MQIA_CURRENT_Q_DEPTH)

/* Send inquiry to the command server and wait for reply */
mqExecute(MQCMD_INQUIRE_Q, ...)
```

For more examples of filtering and querying data items, see "Examples of using the MQAI" on page 20.

Inquiring within data bags

You can inquire about:

- The value of an integer item using the mqInquireInteger call. See mqInquireInteger.
- The value of a 64-bit integer item using the mqInquireInteger64 call. See mqInquireInteger64.
- The value of an integer filter item using the mqInquireIntegerFilter call. See mqInquireIntegerFilter.
- The value of a character-string item using the mqInquireString call. See mqInquireString.
- The value of a string filter item using the mqInquireStringFilter call. See mqInquireStringFilter.
- The value of a byte string item using the mqInquireByteString call. See mqInquireByteString.
- The value of a byte string filter item using the mqInquireByteStringFilter call. See mqInquireByteStringFilter.
- The value of a bag handle using the mqInquireBag call. See mqInquireBag.

You can also inquire about the type (integer, 64-bit integer, integer filter, character string, string filter, byte string, byte string filter or bag handle) of a specific item using the mqInquireItemInfo call. See mqInquireItemInfo.

Changing information within a bag

The MQAI lets you change information within a bag using the mqSet* calls. You can:

1. Modify data items within a bag. The index allows an individual instance of a parameter to be replaced by identifying the occurrence of the item to be modified (see Figure 4 on page 51).

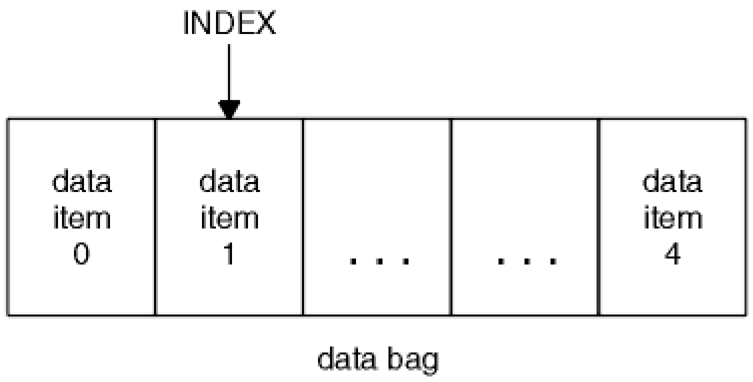


Figure 4. Modifying a single data item

2. Delete all existing occurrences of the specified selector and add a new occurrence to the end of the bag. (See Figure 5 on page 52.) A special index value allows ***all*** instances of a parameter to be replaced.

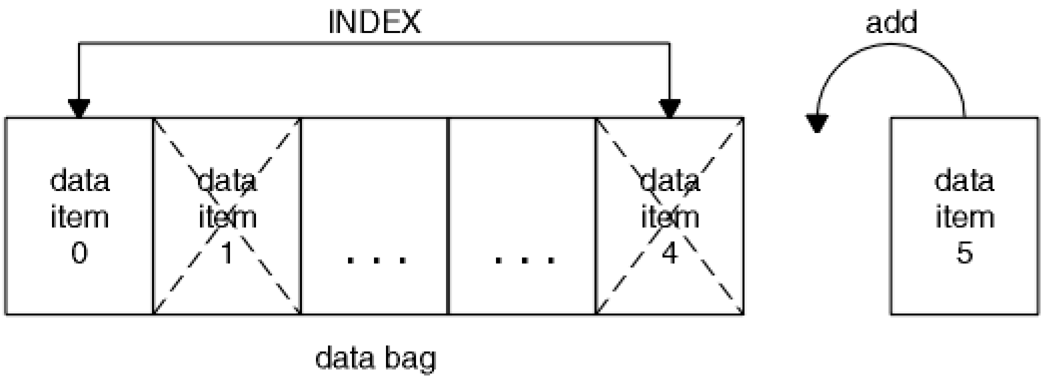


Figure 5. Modifying all data items

Note: The index preserves the insertion order within the bag but can affect the indices of other data items.

The mqSetInteger call lets you modify integer items within a bag. The mqSetInteger64 call lets you modify 64-bit integer items. The mqSetIntegerFilter call lets you modify integer filter items. The mqSetString call lets you modify character-string items. The mqSetStringFilter call lets you modify string filter items. The mqSetByteString call lets you modify byte string items. The mqSetByteStringFilter call lets you modify byte string filter items. Alternatively, you can use these calls to delete all existing occurrences of the specified selector and add a new occurrence at the end of the bag. The data item can be a user item or a system item.

For a full description of these calls, see:

- mqSetInteger
- mqSetInteger64
- mqSetIntegerFilter
- mqSetString
- mqSetStringFilter
- mqSetByteString
- mqSetByteStringFilter

Clearing a bag using the mqClearBag call

The mqClearBag call removes all user items from a user bag and resets system items to their initial values. System bags contained within the bag are also deleted.

For a full description of the mqClearBag call, see mqClearBag.

Truncating a bag using the mqTruncateBag call

The mqTruncateBag call reduces the number of user items in a user bag by deleting the items from the end of the bag, starting with the most recently added item. For example, it can be used when using the same header information to generate more than one message.

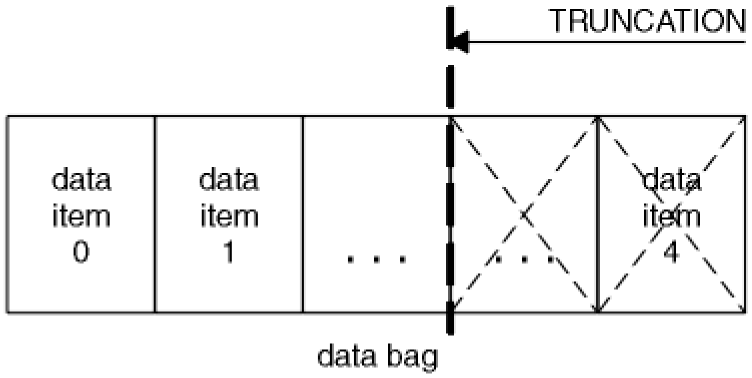


Figure 6. Truncating a bag

For a full description of the mqTruncateBag call, see mqTruncateBag.

Converting bags and buffers

To send data between applications, firstly the message data is placed in a bag. Then, the data in the bag is converted into a PCF message using the mqBagToBuffer call. The PCF message is sent to the required queue using the MQPUT call. This is shown in Figure Figure 7 on page 53. For a full description of the mqBagToBuffer call, see mqBagToBuffer.

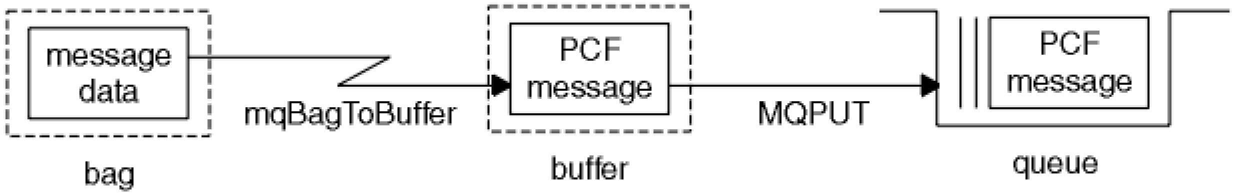


Figure 7. Converting bags to PCF messages

To receive data, the message is received into a buffer using the MQGET call. The data in the buffer is then converted into a bag using the mqBufferToBag call, providing the buffer contains a valid PCF message. This is shown in Figure Figure 8 on page 53. For a full description of the mqBufferToBag call, see mqBufferToBag.

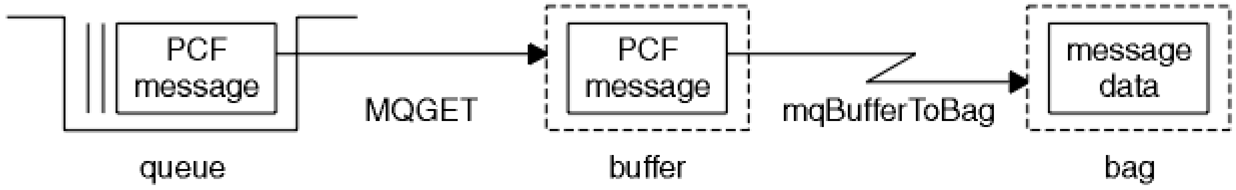


Figure 8. Converting PCF messages to bag form

Counting data items

The mqCountItems call counts the number of user items, system items, or both, that are stored in a data bag, and returns this number. For example, `mqCountItems(`*`Bag`*`, 7, ...)`, returns the number of items in the bag with a selector of 7. It can count items by individual selector, by user selectors, by system selectors, or by all selectors.

Note: This call counts the number of data items, not the number of unique selectors in the bag. A selector can occur multiple times, so there might be fewer unique selectors in the bag than data items.

For a full description of the mqCountItems call, see mqCountItems.

Deleting data items

You can delete items from bags in a number of ways. You can:

- Remove one or more user items from a bag. For detailed information, see "Deleting data items from a bag using the mqDeleteItem call" on page 54.
- Delete ***all*** user items from a bag, that is, *clear* a bag. For detailed information see "Clearing a bag using the mqClearBag call" on page 52.
- Delete user items from the end of a bag, that is, *truncate* a bag. For detailed information, see "Truncating a bag using the mqTruncateBag call" on page 52.

Deleting data items from a bag using the mqDeleteItem call

The mqDeleteItem call removes one or more user items from a bag. The index is used to delete either:

1. A single occurrence of the specified selector. (See Figure 9 on page 54.)

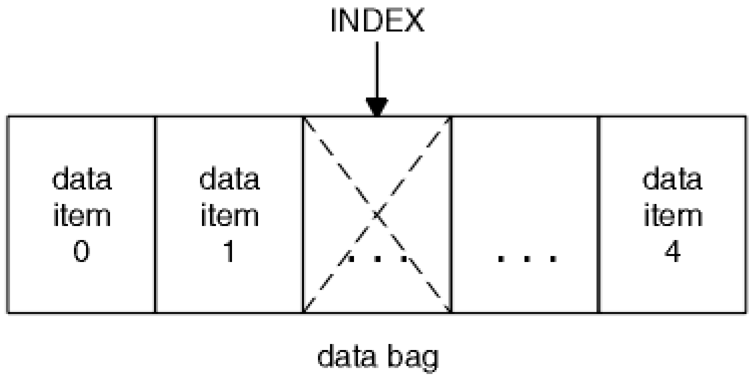


Figure 9. Deleting a single data item

or

2. All occurrences of the specified selector. (See Figure 10 on page 54.)

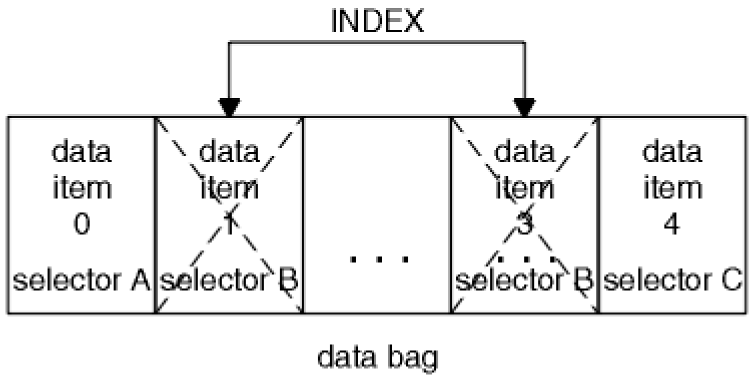


Figure 10. Deleting all data items

Note: The index preserves the insertion order within the bag but can affect the indices of other data items. For example, the mqDeleteItem call does not preserve the index values of the data items that follow the deleted item because the indices are reorganized to fill the gap that remains from the deleted item.

For a full description of the mqDeleteItem call, see mqDeleteItem.

Sending administration commands to the command server using the mqExecute call

When a data bag has been created and populated, an administrative command message can be sent to the command server of a queue manager using the mqExecute call. This handles the exchange with the command server and returns responses in a bag.

After you have created and populated your data bag, you can send an administration command message to the command server of a queue manager. The easiest way to do this is by using the mqExecute call.

The mqExecute call sends an administration command message as a nonpersistent message and waits for any responses. Responses are returned in a response bag. These might contain information about attributes relating to several WebSphere MQ objects or a series of PCF error response messages, for example. Therefore, the response bag could contain a return code only or it could contain *nested bags*.

Response messages are placed into system bags that are created by the system. For example, for inquiries about the names of objects, a system bag is created to hold those object names and the bag is inserted into the user bag. Handles to these bags are then inserted into the response bag and the nested bag can be accessed by the selector MQHA_BAG_HANDLE. The system bag stays in storage, if it is not deleted, until the response bag is deleted.

The concept of *nesting* is shown in Figure 11 on page 55.

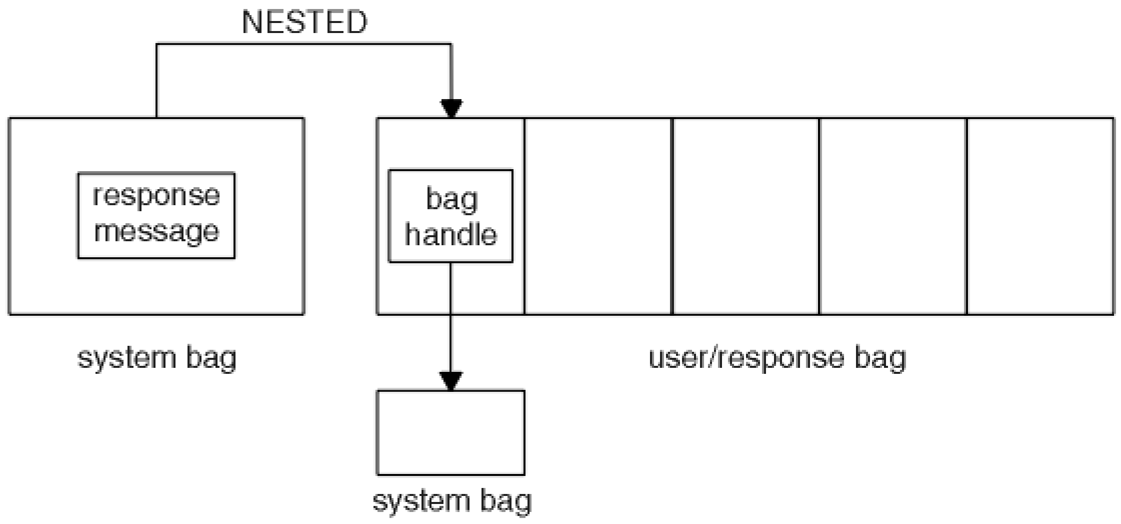


Figure 11. Nesting

As input to the mqExecute call, you must supply:

- An MQI connection handle.
- The command to be executed. This should be one of the MQCMD_* values.

  Note: If this value is not recognized by the MQAI, the value is still accepted. However, if the mqAddInquiry call was used to insert values into the bag, this parameter must be an INQUIRE command recognized by the MQAI. That is, the parameter should be of the form MQCMD_INQUIRE_*.
- Optionally, a handle of the bag containing options that control the processing of the call. This is also where you can specify the maximum time in milliseconds that the MQAI should wait for each reply message.
- A handle of the administration bag that contains details of the administration command to be issued.
- A handle of the response bag that receives the reply messages.

The following are optional:

- An object handle of the queue where the administration command is to be placed.

  If no object handle is specified, the administration command is placed on the SYSTEM.ADMIN.COMMAND.QUEUE belonging to the currently connected queue manager. This is the default.
- An object handle of the queue where reply messages are to be placed.

  You can choose to place the reply messages on a dynamic queue that is created automatically by the MQAI. The queue created exists for the duration of the call only, and is deleted by the MQAI on exit from the mqExecute call.

For examples uses of the mqExecute call, see Example code

Administration using the IBM WebSphere MQ Explorer

The IBM WebSphere MQ Explorer allows you to perform local or remote administration of your network from a computer running Windows, or Linux (x86 and x86-64 platforms) only.

IBM WebSphere MQ for Windows, and IBM WebSphere MQ for Linux (x86 and x86-64 platforms) provide an administration interface called the IBM WebSphere MQ Explorer to perform administration tasks as an alternative to using control or MQSC commands. Comparing command sets shows you what you can do using the IBM WebSphere MQ Explorer.

The IBM WebSphere MQ Explorer allows you to perform local or remote administration of your network from a computer running Windows, or Linux (x86-64 platforms), by pointing the IBM WebSphere MQ Explorer at the queue managers and clusters you are interested in. The platforms and levels of IBM WebSphere MQ that can be administered using the IBM WebSphere MQ Explorer are described in “Remote queue managers” on page 57.

To configure remote IBM WebSphere MQ queue managers so that IBM WebSphere MQ Explorer can administer them, see “Prerequisite software and definitions” on page 58.

It allows you to perform tasks, typically associated with setting up and fine-tuning the working environment for IBM WebSphere MQ, either locally or remotely within a Windows or Linux (x86 and x86-64 platforms) system domain.

On Linux, the IBM WebSphere MQ Explorer might fail to start if you have more than one Eclipse installation. If this happens, start the IBM WebSphere MQ Explorer using a different user ID to the one you use for the other Eclipse installation.

On Linux, to start the IBM WebSphere MQ Explorer successfully, you must be able to write a file to your home directory, and the home directory must exist.

What you can do with the IBM WebSphere MQ Explorer

This is a list of the tasks that you can perform using the IBM WebSphere MQ Explorer.

With the IBM WebSphere MQ Explorer, you can:

- Create and delete a queue manager (on your local machine only).
- Start and stop a queue manager (on your local machine only).
- Define, display, and alter the definitions of WebSphere MQ objects such as queues and channels.
- Browse the messages on a queue.
- Start and stop a channel.
- View status information about a channel, listener, queue, or service objects.
- View queue managers in a cluster.
- Check to see which applications, users, or channels have a particular queue open.
- Create a new queue manager cluster using the *Create New Cluster* wizard.
- Add a queue manager to a cluster using the *Add Queue Manager to Cluster* wizard.
- Manage the authentication information object, used with Secure Sockets Layer (SSL) channel security.
- Create and delete channel initiators, trigger monitors, and listeners.
- Start or stop the command servers, channel initiators, trigger monitors, and listeners.
- Set specific services to start automatically when a queue manager is started.
- Modify the properties of queue managers.
- Change the local default queue manager.
- Invoke the ikeyman GUI to manage secure sockets layer (SSL) certificates, associate certificates with queue managers, and configure and setup certificate stores (on your local machine only).
- Create JMS objects from WebSphere MQ objects, and WebSphere MQ objects from JMS objects.
- Create a JMS Connection Factory for any of the currently supported types.

- Modify the parameters for any service, such as the TCP port number for a listener, or a channel initiator queue name.
- Start or stop the service trace.

You perform administration tasks using a series of *Content Views* and *Property dialogs*.

Content View
A Content View is a panel that can display the following:

- Attributes, and administrative options relating to WebSphere MQ itself.
- Attributes, and administrative options relating to one or more related objects.
- Attributes, and administrative options for a cluster.

Property dialogs
A property dialog is a panel that displays attributes relating to an object in a series of fields, some of which you can edit.

You navigate through the WebSphere MQ Explorer using the *Navigator view*. The Navigator allows you to select the Content View you require.

Remote queue managers

There are two exceptions to the supported queue managers that you can connect to.

From a Windows or Linux (x86 and x86-64 platforms) system, the WebSphere MQ Explorer can connect to all supported queue managers with the following exceptions:

- WebSphere MQ for z/OS queue managers earlier than Version 6.0.
- Currently supported MQSeries® V2 queue managers.

The IBM WebSphere MQ Explorer handles the differences in the capabilities between the different command levels and platforms. However, if it encounters an attribute that it does not recognize, the attribute will not be visible.

If you intend to remotely administer a V6.0 or later queue manager on Windows using the IBM WebSphere MQ Explorer on a WebSphere MQ V5.3 computer, you must install Fix Pack 9 (CSD9) or later on your WebSphere MQ for Windows V5.3 computer.

If you intend to remotely administer a V5.3 queue manager on iSeries using the WebSphere MQ Explorer on a WebSphere MQ V6.0 or later computer, you must install Fix Pack 11 (CSD11) or later on your WebSphere MQ for iSeries V5.3 computer. This fix pack corrects connection problems between the WebSphere MQ Explorer and the iSeries queue manager.

Deciding whether to use the IBM WebSphere MQ Explorer

When deciding whether to use the IBM WebSphere MQ Explorer at your installation, consider the information listed in this topic.

You need to be aware of the following points:

Object names
If you use lowercase names for queue managers and other objects with the IBM WebSphere MQ Explorer, when you work with the objects using MQSC commands, you must enclose the object names in single quotation marks, or WebSphere MQ does not recognize them.

Large queue managers
The IBM WebSphere MQ Explorer works best with small queue managers. If you have a large number of objects on a single queue manager, you might experience delays while the WebSphere MQ Explorer extracts the required information to present in a view.

Clusters
WebSphere MQ clusters can potentially contain hundreds or thousands of queue managers. The WebSphere MQ Explorer presents the queue managers in a cluster using a tree structure. The physical size of a cluster does not affect the speed of the IBM WebSphere MQ Explorer dramatically because

the IBM WebSphere MQ Explorer does not connect to the queue managers in the cluster until you select them.

Setting up the IBM WebSphere MQ Explorer

This section outlines the steps you need to take to set up the IBM WebSphere MQ Explorer.

- "Prerequisite software and definitions" on page 58
- "Security" on page 58
- "Showing and hiding queue managers and clusters" on page 62
- "Cluster membership" on page 62
- "Data conversion" on page 63

Prerequisite software and definitions

Ensure that you satisfy the following requirements before trying to use the IBM WebSphere MQ Explorer.

The IBM WebSphere MQ Explorer can connect to remote queue managers using the TCP/IP communication protocol only.

Check that:

1. A command server is running on every remotely administered queue manager.
2. A suitable TCP/IP listener object must be running on every remote queue manager. This object can be the IBM WebSphere MQ listener or, on UNIX and Linux systems, the inetd daemon.
3. A server-connection channel, by default named SYSTEM.ADMIN.SVRCONN, exists on all remote queue managers.

   You can create the channel using the following MQSC command:

   ```
   DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN)
   ```

   This command creates a basic channel definition. If you want a more sophisticated definition (to set up security, for example), you need additional parameters. For more information, see DEFINE CHANNEL .
4. The system queue, SYSTEM.MQEXPLORER.REPLY.MODEL, must exist.

Security

If you are using WebSphere MQ in an environment where it is important for you to control user access to particular objects, you might need to consider the security aspects of using the IBM WebSphere MQ Explorer.

Authorization to use the IBM WebSphere MQ Explorer

Any user can use the IBM WebSphere MQ Explorer, but certain authorities are required to connect, access, and manage queue managers.

To perform local administrative tasks using the WebSphere MQ Explorer, a user is required to have the necessary authority to perform the administrative tasks. If the user is a member of the mqm group, the user has authority to perform all local administrative tasks.

To connect to a remote queue manager and perform remote administrative tasks using the WebSphere MQ Explorer, the user executing the WebSphere MQ Explorer is required to have the following authorities:

- CONNECT authority on the target queue manager object
- INQUIRE authority on the target queue manager object
- DISPLAY authority to the target queue manager object
- INQUIRE authority to the queue, SYSTEM.MQEXPLORER.REPLY.MODEL
- DISPLAY authority to the queue, SYSTEM.MQEXPLORER.REPLY.MODEL

- INPUT (get) authority to the queue, SYSTEM.MQEXPLORER.REPLY.MODEL
- OUTPUT (put) authority to the queue, SYSTEM.ADMIN.COMMAND.QUEUE
- INQUIRE authority on the queue, SYSTEM.ADMIN.COMMAND.QUEUE
- Authority to perform the action selected

Note: INPUT authority relates to input to the user from a queue (a get operation). OUTPUT authority relates to output from the user to a queue (a put operation).

To connect to a remote queue manager on WebSphere MQ for z/OS and perform remote administrative tasks using the IBM WebSphere MQ Explorer, the following must be provided:

- A RACF® profile for the system queue, SYSTEM.MQEXPLORER.REPLY.MODEL
- A RACF profile for the queues, AMQ.MQEXPLORER.*

In addition, the user executing the WebSphere MQ Explorer is required to have the following authorities:

- RACF UPDATE authority to the system queue, SYSTEM.MQEXPLORER.REPLY.MODEL
- RACF UPDATE authority to the queues, AMQ.MQEXPLORER.*
- CONNECT authority on the target queue manager object
- Authority to perform the action selected
- READ authority to all the hlq.DISPLAY.object profiles in the MQCMDS class

For information about how to grant authority to WebSphere MQ objects, see Giving access to a WebSphere MQ object on UNIX or Linux systems and Windows.

If a user attempts to perform an operation that they are not authorized to perform, the target queue manager invokes authorization failure procedures and the operation fails.

The default filter in the WebSphere MQ Explorer is to display all WebSphere MQ objects. If there are any WebSphere MQ objects that a user does not have DISPLAY authority to, authorization failures are generated. If authority events are being recorded, restrict the range of objects that are displayed to those objects that the user has DISPLAY authority to.

Security for connecting to remote queue managers

You must secure the channel between the IBM WebSphere MQ Explorer and each remote queue manager.

The IBM WebSphere MQ Explorer connects to remote queue managers as an MQI client application. This means that each remote queue manager must have a definition of a server-connection channel and a suitable TCP/IP listener. If you do not secure your server connection channel it is possible for a malicious application to connect to the same server connection channel and gain access to the queue manager objects with unlimited authority. In order to secure your server connection channel either specify a non-blank value for the MCAUSER attribute of the channel, use channel authentication records, or use a security exit.

The default value of the MCAUSER attribute is the local user ID. If you specify a non-blank user name as the MCAUSER attribute of the server connection channel, all programs connecting to the queue manager using this channel run with the identity of the named user and have the same level of authority. This does not happen if you use channel authentication records.

Using a security exit with the WebSphere MQ Explorer

You can specify a default security exit and queue manager specific security exits using the WebSphere MQ Explorer.

You can define a default security exit, which can be used for all new client connections from the WebSphere MQ Explorer. This default exit can be overridden at the time a connection is made. You can also define a security exit for a single queue manager or a set of queue managers, which takes effect when a connection is made. You specify exits using the WebSphere MQ Explorer. For more information, see the WebSphere MQ Help Center.

Using the IBM WebSphere MQ Explorer to connect to a remote queue manager using SSL-enabled MQI channels

The IBM WebSphere MQ Explorer connects to remote queue managers using an MQI channel. If you want to secure the MQI channel using SSL security, you must establish the channel using a client channel definition table.

For information how to establish an MQI channel using a client channel definition table, see Overview of IBM WebSphere MQ MQI clients.

When you have established the channel using a client channel definition table, you can use the IBM WebSphere MQ Explorer to connect to a remote queue manager using SSL-enabled MQI channel, as described in "Tasks on the system that hosts the remote queue manager" on page 60 and "Tasks on the system that hosts the IBM WebSphere MQ Explorer" on page 60.

Tasks on the system that hosts the remote queue manager

On the system hosting the remote queue manager, perform the following tasks:

1. Define a server connection and client connection pair of channels, and specify the appropriate value for the *SSLCIPH* variable on the server connection on both channels. For more information about the *SSLCIPH* variable, see Protecting channels with SSL
2. Send the channel definition table `AMQCLCHL.TAB` , which is found in the queue manager's `@ipcc` directory, to the system hosting the IBM WebSphere MQ Explorer.
3. Start a TCP/IP listener on a designated port.
4. Place both the CA and personal SSL certificates into the SSL directory of the queue manager:
   - `/var/mqm/qmgrs/+QMNAME+/SSL` for UNIX and Linux systems
   - `C:\Program Files\WebSphere MQ\qmgrs\+QMNAME+\SSL` for Windows systems

   Where +QMNAME+ is a token representing the name of the queue manager.
5. Create a key database file of type CMS named `key.kdb` . Stash the password in a file either by checking the option in the iKeyman GUI, or by using the `-stash` option with the **`runmqckm`** commands.
6. Add the CA certificates to the key database created in the previous step.
7. Import the personal certificate for the queue manager into the key database.

For more detailed information about working with the Secure Sockets Layer on Windows systems, see Working with SSL or TLS on UNIX, Linux and Windows systems .

Tasks on the system that hosts the IBM WebSphere MQ Explorer

On the system hosting the IBM WebSphere MQ Explorer, perform the following tasks:

1. Create a key database file of type JKS named `key.jks`. Set a password for this key database file.

   The IBM WebSphere MQ Explorer uses Java keystore files (JKS) for SSL security, and so the keystore file being created for configuring SSL for the IBM WebSphere MQ Explorer must match this.
2. Add the CA certificates to the key database created in the previous step.
3. Import the personal certificate for the queue manager into the key database.
4. On Windows and Linux systems, start MQ Explorer by using the system menu, the `MQExplorer` executable file, or the **`strmqcfg`** command.
5. From the IBM WebSphere MQ Explorer toolbar, click **Window -> Preferences**, then expand **WebSphere MQ Explorer** and click **SSL Client Certificate Stores**. Enter the name of, and password for, the JKS file created in step 1 of "Tasks on the system that hosts the IBM WebSphere MQ Explorer" on page 60, in both the Trusted Certificate Store and the Personal Certificate Store, then click **OK**.
6. Close the **Preferences** window, and right-click **Queue Managers**. Click **Show/Hide Queue Managers**, and then click **Add** on the **Show/Hide Queue Managers** screen.

7. Type the name of the queue manager, and select the **Connect directly** option. Click next.
8. Select **Use client channel definition table (CCDT)** and specify the location of the channel table file that you transferred from the remote queue manager in step 2 in "Tasks on the system that hosts the remote queue manager" on page 60 on the system hosting the remote queue manager.
9. Click **Finish**. You can now access the remote queue manager from the IBM WebSphere MQ Explorer.

Connecting through another queue manager

The IBM WebSphere MQ Explorer allows you to connect to a queue manager through an intermediate queue manager, to which the IBM WebSphere MQ Explorer is already connected.

In this case, the IBM WebSphere MQ Explorer puts PCF command messages to the intermediate queue manager, specifying the following:

- The *ObjectQMgrName* parameter in the object descriptor (MQOD) as the name of the target queue manager. For more information on queue name resolution, see the Name resolution .
- The *UserIdentifier* parameter in the message descriptor (MQMD) as the local userId.

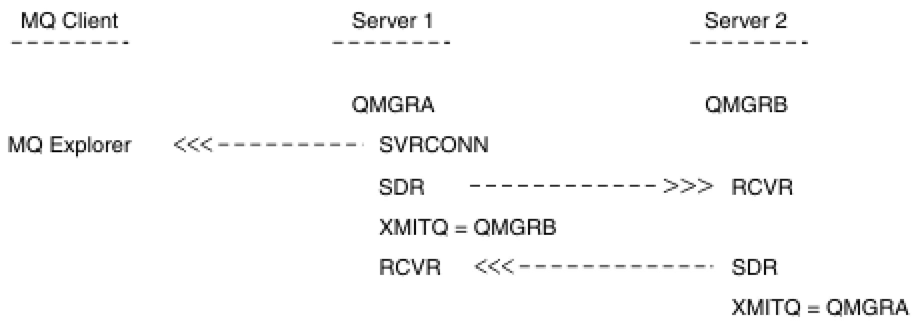
If the connection is then used to connect to the target queue manager via an intermediate queue manager, the userId is flowed in the *UserIdentifier* parameter of the message descriptor (MQMD) again. In order for the MCA listener on the target queue manager to accept this message, either the MCAUSER attribute must be set, or the userId must already exist with put authority.

The command server on the target queue manager puts messages to the transmission queue specifying the userId in the *UserIdentifier* parameter in the message descriptor (MQMD). For this put to succeed the userId must already exist on the target queue manager with put authority.

The following example shows you how to connect a queue manager, through an intermediate queue manager, to the WebSphere MQ Explorer.

Establish a remote administration connection to a queue manager. Verify that the:

- Queue manager on the server is active and has a server-connection channel (SVRCONN) defined.
- Listener is active.
- Command server is active.
- SYSTEM.MQ EXPLORER.REPLY.MODEL queue has been created and that you have sufficient authority.
- Queue manager listeners, command servers, and sender channels are started.



In this example:

- IBM WebSphere MQ Explorer is connected to queue manager QMGRA (running on Server1) using a client connection.
- Queue manager QMGRB on Server2 can be now connected to IBM WebSphere MQ Explorer through an intermediate queue manager (QMGRA)
- When connecting to QMGRB with WebSphere MQ Explorer, select QMGRA as the intermediate queue manager

In this situation, there is no direct connection to QMGRB from IBM WebSphere MQ Explorer; the connection to QMGRB is through QMGRA.

Queue manager QMGRB on Server2 is connected to QMGRA on Server1 using sender-receiver channels. The channel between QMGRA and QMGRB must be set up in such a way that remote administration is possible; see "Preparing channels and transmission queues for remote administration" on page 105.

Showing and hiding queue managers and clusters

The IBM WebSphere MQ Explorer can display more than one queue manager at a time. From the Show/Hide Queue Manager panel (selectable from the menu for the Queue Managers tree node), you can choose whether you display information about another (remote) machine. Local queue managers are detected automatically.

To show a remote queue manager:

1. Right-click the `Queue Managers` tree node, then select *Show/Hide Queue Managers...*.
2. Click **Add**. The Show/Hide Queue Managers panel is displayed.
3. Enter the name of the remote queue manager and the host name or IP address in the fields provided.

   The host name or IP address is used to establish a client connection to the remote queue manager using either its default server connection channel, SYSTEM.ADMIN.SVRCONN, or a user-defined server connection channel.
4. Click **Finish**.

The Show/Hide Queue Managers panel also displays a list of all visible queue managers. You can use this panel to hide queue managers from the navigation view.

If the IBM WebSphere MQ Explorer displays a queue manager that is a member of a cluster, the cluster is detected, and displayed automatically.

To export the list of remote queue managers from this panel:

1. Close the Show/Hide Queue Managers panel.
2. Right-click the top **IBM WebSphere MQ** tree node in the Navigation pane of the WebSphere MQ Explorer, then select **Export MQ Explorer Settings**
3. Click **MQ Explorer > MQ Explorer Settings**
4. Select **Connection Information > Remote queue managers**.
5. Select a file to store the exported settings in.
6. Finally, click **Finish** to export the remote queue manager connection information to the specified file.

To import a list of remote queue managers:

1. Right-click the top **IBM WebSphere MQ** tree node in the Navigation pane of the WebSphere MQ Explorer, then select **Import MQ Explorer Settings**
2. Click **MQ Explorer > MQ Explorer Settings**
3. Click **Browse**, and navigate to the path of the file that contains the remote queue manager connection information.
4. Click **Open**. If the file contains a list of remote queue managers, the **Connection Information > Remote queue managers** box is selected.
5. Finally, click **Finish** to import the remote queue manager connection information into the WebSphere MQ Explorer.

Cluster membership

IBM WebSphere MQ Explorer requires information about queue managers that are members of a cluster.

If a queue manager is a member of a cluster, then the cluster tree node will be populated automatically.

If queue managers become members of clusters while the IBM WebSphere MQ Explorer is running, then you must maintain the IBM WebSphere MQ Explorer with up-to-date administration data about clusters so that it can communicate effectively with them and display correct cluster information when requested. In order to do this, the WebSphere MQ Explorer needs the following information:

- The name of a repository queue manager
- The connection name of the repository queue manager if it is on a remote queue manager

With this information, the WebSphere MQ Explorer can:

- Use the repository queue manager to obtain a list of queue managers in the cluster.
- Administer the queue managers that are members of the cluster and are on supported platforms and command levels.

Administration is not possible if:

- The chosen repository becomes unavailable. The WebSphere MQ Explorer does not automatically switch to an alternative repository.
- The chosen repository cannot be contacted over TCP/IP.
- The chosen repository is running on a queue manager that is running on a platform and command level not supported by the WebSphere MQ Explorer.

The cluster members that can be administered can be local, or they can be remote if they can be contacted using TCP/IP. The IBM WebSphere MQ Explorer connects to local queue managers that are members of a cluster directly, without using a client connection.

Data conversion

The IBM WebSphere MQ Explorer works in CCSID 1208 (UTF-8). This enables the IBM WebSphere MQ Explorer to display the data from remote queue managers correctly. Whether connecting to a queue manager directly, or by using an intermediate queue manager, the IBM WebSphere MQ Explorer requires all incoming messages to be converted to CCSID 1208 (UTF-8).

An error message is issued if you try to establish a connection between the IBM WebSphere MQ Explorer and a queue manager with a CCSID that the IBM WebSphere MQ Explorer does not recognize.

Supported conversions are described in Code page conversion.

Security on Windows

The Prepare WebSphere MQ wizard creates a special user account so that the Windows service can be shared by processes that need to use it.

A Windows service is shared between client processes for an IBM WebSphere MQ installation. One service is created for each installation. Each service is named `MQ_`*`InstallationName`*, and has a display name of `IBM WebSphere MQ(`*`InstallationName`*`)`. Before Version 7.1, with only one installation on a server the single, Windows service was named `MQSeriesServices` with the display name `IBM MQSeries`.

Because each service must be shared between non-interactive and interactive logon sessions, you must launch each under a special user account. You can use one special user account for all the services, or create different special user accounts. Each special user account must have the user right to "Logon as a service", for more information see "User rights required for an IBM WebSphere MQ Windows Service" on page 64. If the user ID does not have the authority to run the service, the service does not start and it returns an error in the Windows system event log. Typically, you will have run the Prepare IBM WebSphere MQ wizard, and set up the user ID correctly. However, if you have configured the user ID manually, is it possible that you might have a problem that you will need to resolve.

When you install IBM WebSphere MQ and run the Prepare IBM WebSphere MQ wizard for the first time, it creates a local user account for the service called MUSR_MQADMIN with the required settings and permissions, including "Logon as a service".

For subsequent installations, the Prepare IBM WebSphere MQ wizard creates a user account named MUSR_MQADMINx, where x is the next available number representing a user ID that does not exist. The password for MUSR_MQADMINx is randomly generated when the account is created, and used to configure the logon environment for the service. The generated password does not expire.

This IBM WebSphere MQ account is not affected by any account policies that are set up on the system to require that account passwords are changed after a certain period.

The password is not known outside this one-time processing and is stored by the Windows operating system in a secure part of the registry.

Using Active directory (Windows only)

In some network configurations, where user accounts are defined on domain controllers that are using Active Directory, the local user account IBM WebSphere MQ is running under might not have the authority it requires to query the group membership of other domain user accounts. The Prepare IBM WebSphere MQ Wizard identifies whether this is the case by carrying out tests and asking the user questions about the network configuration.

If the local user account IBM WebSphere MQ is running under does not have the required authority, the Prepare IBM WebSphere MQ Wizard prompts the user for the account details of a domain user account with particular user rights. For the user rights that the domain user account requires see "User rights required for an IBM WebSphere MQ Windows Service" on page 64. Once the user has entered valid account details for the domain user account into the Prepare IBM WebSphere MQ Wizard, it configures an IBM WebSphere MQ Windows service to run under the new account. The account details are held in the secure part of the Registry and cannot be read by users.

When the service is running, an IBM WebSphere MQ Windows service is launched and remains running for as long as the service is running. An IBM WebSphere MQ administrator who logs on to the server after the Windows service is launched can use the IBM WebSphere MQ Explorer to administer queue managers on the server. This connects the IBM WebSphere MQ Explorer to the existing Windows service process. These two actions need different levels of permission before they can work:

- The launch process requires a launch permission.
- The IBM WebSphere MQ administrator requires Access permission.

User rights required for an IBM WebSphere MQ Windows Service

The table in this topic lists the user rights required for the local and domain user account under which the Windows service for an IBM WebSphere MQ installation runs.

Log on as batch job	Enables an IBM WebSphere MQ Windows service to run under this user account.
Log on as service	Enables users to set the IBM WebSphere MQ Windows service to log on using the configured account.
Shut down the system	Allows the IBM WebSphere MQ Windows service to restart the server if configured to do so when recovery of a service fails.
Increase quotas	Required for operating system `CreateProcessAsUser` call.
Act as part of the operating system	Required for operating system `LogonUser` call.
Bypass traverse checking	Required for operating system `LogonUser` call.
Replace a process level token	Required for operating system `LogonUser` call.

Note: Debug programs rights might be needed in environments running ASP and IIS applications.

Your domain user account must have these Windows user rights set as effective user rights as listed in the Local Security Policy application. If they are not, set them using either the Local Security Policy application locally on the server, or by using the Domain Security Application domain wide.

Changing the user name associated with the IBM WebSphere MQ Service

You might need to change the user name associated with the IBM WebSphere MQ Service from MUSR_MQADMIN to something else. (For example, you might need to do this if your queue manager is associated with DB2®, which does not accept user names of more than 8 characters.)

Procedure

1. Create a new user account (for example **NEW_NAME**)
2. Use the Prepare IBM WebSphere MQ Wizard to enter the details of the new user account.

Changing the password of the IBM WebSphere MQ Windows service user account

About this task

To change the password of theIBM WebSphere MQ Windows service local user account, perform the following steps:

Procedure

1. Identify the user the service is running under.
2. Stop the IBMIBM WebSphere MQ service from the Computer Management panel.
3. Change the required password in the same way that you would change the password of an individual.
4. Go to the properties for the IBM WebSphere MQ service from the Computer Management panel.
5. Select the **Log On** Page.
6. Confirm that the account name specified matches the user for which the password was modified.
7. Type the password into the **Password** and **Confirm password** fields and click **OK**.

IBM WebSphere MQ Windows service for an installation running under a domain user account

About this task

If the IBM WebSphere MQ Windows service for an installation is running under a domain user account, you can also change the password for the account as follows:

Procedure

1. Change the password for the domain account on the domain controller. You might need to ask your domain administrator to do this for you.
2. Follow the steps to modify the **Log On** page for the IBMIBM WebSphere MQ service.

   The user account that IBM WebSphere MQ Windows service runs under executes any MQSC commands that are issued by user interface applications, or performed automatically on system startup, shutdown, or service recovery. This user account must therefore have IBM WebSphere MQ administration rights. By default it is added to the local **mqm** group on the server. If this membership is removed, the IBM WebSphere MQ Windows service does not work. For more information about user rights, see "User rights required for an IBM WebSphere MQ Windows Service" on page 64

   If a security problem arises with the user account that the IBM WebSphere MQ Windows service runs under, error messages and descriptions appear in the system event log.

Related concepts

"Using Active directory (Windows only)" on page 64
In some network configurations, where user accounts are defined on domain controllers that are using Active Directory, the local user account IBM WebSphere MQ is running under might not have the authority

it requires to query the group membership of other domain user accounts. The Prepare IBM WebSphere MQ Wizard identifies whether this is the case by carrying out tests and asking the user questions about the network configuration.

IBM WebSphere MQ coordinating with Db2 as the resource manager

If you start your queue managers from the IBM WebSphere MQ Explorer, or are using IBM WebSphere MQ V7, and are having problems when coordinating Db2, check your queue manager error logs.

Check your queue manager error logs for an error like the following:

```
23/09/2008 15:43:54 - Process(5508.1) User(MUSR_MQADMIN) Program(amqzxma0.exe)
Host(HOST_1) Installation(Installation1)
VMRF(7.1.0.0) QMgr(A.B.C)
AMQ7604: The XA resource manager 'DB2 MQBankDB database' was not available when called
    for xa_open. The queue manager is continuing without this resource manager.
```

Explanation: The user ID (default name is MUSR_MQADMIN) which runs theIBM WebSphere MQ Service process `amqsvc.exe` is still running with an access token which does not contain group membership information for the group DB2USERS.

Solve: After you have ensured that the IBM WebSphere MQ Service user ID is a member of DB2USERS, use the following sequence of commands:

- stop the service.
- stop any other processes running under the same user ID.
- restart these processes.

Rebooting the machine would ensure the previous steps, but is not necessary.

Extending the IBM WebSphere MQ Explorer

IBM WebSphere MQ for Windows, and IBM WebSphere MQ for Linux (x86 and x86-64 platforms) provide an administration interface called the IBM WebSphere MQ Explorer to perform administration tasks as an alternative to using control or MQSC commands.

This information applies to WebSphere MQ for Windows, and WebSphere MQ for Linux (x86 and x86-64 platforms) only.

The IBM WebSphere MQ Explorer presents information in a style consistent with that of the Eclipse framework and the other plug-in applications that Eclipse supports.

Through extending the IBM WebSphere MQ Explorer, system administrators have the ability to customize the WebSphere MQ Explorer to improve the way they administer WebSphere MQ.

For more information, see *Extending the IBM WebSphere MQ Explorer* in the IBM WebSphere MQ Explorer product documentation.

Using the IBM WebSphere MQ Taskbar application (Windows only)

The IBM WebSphere MQ Taskbar application displays an icon in the Windows system tray on the server. The icon provides you with the current status of IBM WebSphere MQ and a menu from which you can perform some simple actions.

On Windows, the WebSphere MQ icon is in the system tray on the server and is overlaid with a color-coded status symbol, which can have one of the following meanings:

Green
Working correctly; no alerts at present

Blue
Indeterminate; WebSphere MQ is starting up or shutting down

Yellow
Alert; one or more services are failing or have already failed

To display the menu, right-click the WebSphere MQ icon. From the menu you can perform the following actions:

- Click **Open** to open the WebSphere MQ Alert Monitor
- Click **Exit** to exit the WebSphere MQ Taskbar application
- Click **WebSphere MQ Explorer** to start the IBM WebSphere MQ Explorer
- Click **Stop WebSphere MQ** to stop WebSphere MQ
- Click **About WebSphere MQ** to display information about the WebSphere MQ Alert Monitor

The IBM WebSphere MQ alert monitor application (Windows only)

The IBM WebSphere MQ alert monitor is an error detection tool that identifies and records problems with IBM WebSphere MQ on a local machine.

The alert monitor displays information about the current status of the local installation of a WebSphere MQ server. It also monitors the Windows Advanced Configuration and Power Interface (ACPI) and ensures the ACPI settings are enforced.

From the WebSphere MQ alert monitor, you can:

- Access the IBM WebSphere MQ Explorer directly
- View information relating to all outstanding alerts
- Shut down the WebSphere MQ service on the local machine
- Route alert messages over the network to a configurable user account, or to a Windows workstation or server

Administering local IBM WebSphere MQ objects

This section tells you how to administer local IBM WebSphere MQ objects to support application programs that use the Message Queue Interface (MQI). In this context, local administration means creating, displaying, changing, copying, and deleting IBM WebSphere MQ objects.

In addition to the approaches detailed in this section you can use the IBM WebSphere MQ Explorer to administer local WebSphere MQ objects; see "Administration using the IBM WebSphere MQ Explorer" on page 56.

This section contains the following information:

- Application programs using the MQI
- "Performing local administration tasks using MQSC commands" on page 71
- "Working with queue managers" on page 79
- "Working with local queues" on page 81
- "Working with alias queues" on page 86
- "Working with model queues" on page 87
- "Working with services" on page 94
- "Managing objects for triggering" on page 101

Starting and stopping a queue manager

Use this topic as an introduction to stopping and starting a queue manager.

Starting a queue manager

To start a queue manager, use the `strmqm` command as follows:

```
strmqm saturn.queue.manager
```

On WebSphere MQ for Windows and WebSphere MQ for Linux (x86 and x86-64 platforms) systems, you can start a queue manager as follows:

1. Open the IBM WebSphere MQ Explorer.
2. Select the queue manager from the Navigator View.
3. Click `Start`. The queue manager starts.

If the queue manager start-up takes more than a few seconds WebSphere MQ issues information messages intermittently detailing the start-up progress.

The `strmqm` command does not return control until the queue manager has started and is ready to accept connection requests.

Starting a queue manager automatically

In WebSphere MQ for Windows you can start a queue manager automatically when the system starts using the WebSphere MQ Explorer. For more information, see "Administration using the IBM WebSphere MQ Explorer" on page 56.

Stopping a queue manager

Use the **endmqm** command to stop a queue manager.

Note: You must use the **endmqm** command from the installation associated with the queue manager that you are working with. You can find out which installation a queue manager is associated with using the `dspmq -o installation` command.

For example, to stop a queue manager called QMB, enter the following command:

```
endmqm QMB
```

On WebSphere MQ for Windows and WebSphere MQ for Linux (x86 and x86-64 platforms) systems, you can stop a queue manager as follows:

1. Open the IBM WebSphere MQ Explorer.
2. Select the queue manager from the Navigator View.
3. Click `Stop...`. The End Queue Manager panel is displayed.
4. Select Controlled, or Immediate.
5. Click OK. The queue manager stops.

Quiesced shutdown

By default, the **endmqm** command performs a quiesced shutdown of the specified queue manager. This might take a while to complete. A quiesced shutdown waits until all connected applications have disconnected.

Use this type of shutdown to notify applications to stop. If you issue:

```
endmqm -c QMB
```

you are not told when all applications have stopped. (An `endmqm -c QMB` command is equivalent to an `endmqm QMB` command.)

However, if you issue:

```
endmqm -w QMB
```

the command waits until all applications have stopped and the queue manager has ended.

Immediate shutdown

For an immediate shutdown any current MQI calls are allowed to complete, but any new calls fail. This type of shutdown does not wait for applications to disconnect from the queue manager.

For an immediate shutdown, type:

```
endmqm -i QMB
```

Preemptive shutdown

Note: Do not use this method unless all other attempts to stop the queue manager using the **endmqm** command have failed. This method can have unpredictable consequences for connected applications.

If an immediate shutdown does not work, you must resort to a *preemptive* shutdown, specifying the `-p` flag. For example:

```
endmqm -p QMB
```

This stops the queue manager immediately. If this method still does not work, see "Stopping a queue manager manually" on page 69 for an alternative solution.

For a detailed description of the **endmqm** command and its options, see endmqm.

If you have problems shutting down a queue manager

Problems in shutting down a queue manager are often caused by applications. For example, when applications:

- Do not check MQI return codes properly
- Do not request notification of a quiesce
- Terminate without disconnecting from the queue manager (by issuing an `MQDISC` call)

If a problem occurs when you stop the queue manager, you can break out of the **endmqm** command using Ctrl-C. You can then issue another **endmqm** command, but this time with a flag that specifies the type of shutdown that you require.

Stopping a queue manager manually

If the standard methods for stopping queue managers fail, try the methods described here.

The standard way of stopping queue managers is by using the `endmqm` command. To stop a queue manager manually, use one of the procedures described in this section. For details of how to perform operations on queue managers using control commands, see Creating and managing queue managers.

Stopping queue managers on Windows

How to end the processes and the IBM WebSphere MQ service, to stop queue managers in IBM WebSphere MQ for Windows.

To stop a queue manager running under WebSphere MQ for Windows:

1. List the names (IDs) of the processes that are running, by using the Windows Task Manager.
2. End the processes by using Windows Task Manager, or the **`taskkill`** command, in the following order (if they are running):

AMQZMUC0	Critical process manager
AMQZXMA0	Execution controller
AMQZFUMA	OAM process
AMQZLAA0	LQM agents

AMQZLSA0	LQM agents
AMQZMUF0	Utility Manager
AMQZMGR0	Process controller
AMQZMUR0	Restartable process manager
AMQFQPUB	Publish Subscribe process
AMQFCXBA	Broker worker process
AMQRMPPA	Process pooling process
AMQCRSTA	Non-threaded responder job process
AMQCRS6B	LU62 receiver channel and client connection
AMQRRMFA	The repository process (for clusters)
AMQZDMAA	Deferred message processor
AMQPCSEA	The command server
RUNMQTRM	Invoke a trigger monitor for a server
RUNMQDLQ	Invoke dead-letter queue handler
RUNMQCHI	The channel initiator process
RUNMQLSR	The channel listener process
AMQXSSVN	Shared memory servers
AMQZTRCN	Trace

3. Stop the WebSphere MQ service from **Administration tools** > **Services** on the Windows Control Panel.
4. If you have tried all methods and the queue manager has not stopped, reboot your system.

The Windows Task Manager and the **`tasklist`** command give limited information about tasks. For more information to help to determine which processes relate to a particular queue manager, consider using a tool such as *Process Explorer* (procexp.exe), available for download from the Microsoft website at https://www.microsoft.com.

Stopping queue managers on UNIX and Linux systems

How to end the processes and the IBM WebSphere MQ service, to stop queue managers in IBM WebSphere MQ for UNIX and Linux. You can try the methods described here if the standard methods for stopping and removing queue managers fail.

To stop a queue manager running under WebSphere MQ for UNIX and Linux systems:

1. Find the process IDs of the queue manager programs that are still running by using the `ps` command. For example, if the queue manager is called `QMNAME`, use the following command:

   ```
   ps -ef | grep QMNAME
   ```

2. End any queue manager processes that are still running. Use the `kill` command, specifying the process IDs discovered by using the `ps` command.

   End the processes in the following order:

amqzmuc0	Critical process manager
amqzxma0	Execution controller
amqzfuma	OAM process
amqzlaa0	LQM agents
amqzlsa0	LQM agents

amqzmuf0	Utility Manager
amqzmur0	Restartable process manager
amqzmgr0	Process controller
amqfqpub	Publish Subscribe process
amqfcxba	Broker worker process
amqrmppa	Process pooling process
amqcrsta	Non-threaded responder job process
amqcrs6b	LU62 receiver channel and client connection
amqrrmfa	The repository process (for clusters)
amqzdmaa	Deferred message processor
amqpcsea	The command server
runmqtrm	Invoke a trigger monitor for a server
runmqdlq	Invoke dead-letter queue handler
runmqchi	The channel initiator process
runmqlsr	The channel listener process

Note: You can use the **`kill -9`** command to end processes that fail to stop.

If you stop the queue manager manually, FFSTs might be taken, and FDC files placed in `/var/mqm/errors`. Do not regard this as a defect in the queue manager.

The queue manager will restart normally, even after you have stopped it using this method.

Performing local administration tasks using MQSC commands

This section introduces you to MQSC commands and tells you how to use them for some common tasks.

If you use IBM WebSphere MQ for Windows or IBM WebSphere MQ for Linux (x86 and x86-64 platforms), you can also perform the operations described in this section using the IBM WebSphere MQ Explorer. See “Administration using the IBM WebSphere MQ Explorer” on page 56 for more information.

You can use MQSC commands to manage queue manager objects, including the queue manager itself, queues, process definitions, channels, client connection channels, listeners, services, namelists, clusters, and authentication information objects. This section deals with queue managers, queues, and process definitions; for information about administering channel, client connection channel, and listener objects, see Objects . For information about all the MQSC commands for managing queue manager objects, see “Script (MQSC) Commands” on page 72.

You issue MQSC commands to a queue manager using the `runmqsc` command. (For details of this command, see runmqsc.) You can do this interactively, issuing commands from a keyboard, or you can redirect the standard input device (`stdin`) to run a sequence of commands from an ASCII text file. In both cases, the format of the commands is the same. (For information about running the commands from a text file, see “Running MQSC commands from text files” on page 75.)

You can run the `runmqsc` command in three ways, depending on the flags set on the command:

- Verify a command without running it, where the MQSC commands are verified on a local queue manager, but are not run.
- Run a command on a local queue manager, where the MQSC commands are run on a local queue manager.
- Run a command on a remote queue manager, where the MQSC commands are run on a remote queue manager.

You can also run the command followed by a question mark to display the syntax.

Object attributes specified in MQSC commands are shown in this section in uppercase (for example, RQMNAME), although they are not case-sensitive. MQSC command attribute names are limited to eight characters. MQSC commands are available on other platforms, including IBM i and z/OS.

MQSC commands are summarized in the collection of topics in the MQSC reference section.

Script (MQSC) Commands

MQSC commands provide a uniform method of issuing human-readable commands on WebSphere MQ platforms. For information about *programmable command format* (PCF) commands, see "Introduction to Programmable Command Formats" on page 9.

The general format of the commands is shown in The MQSC commands.

You should observe the following rules when using MQSC commands:

- Each command starts with a primary parameter (a verb), and this is followed by a secondary parameter (a noun). This is then followed by the name or generic name of the object (in parentheses) if there is one, which there is on most commands. Following that, parameters can usually occur in any order; if a parameter has a corresponding value, the value must occur directly after the parameter to which it relates.
- Keywords, parentheses, and values can be separated by any number of blanks and commas. A comma shown in the syntax diagrams can always be replaced by one or more blanks. There must be at least one blank immediately preceding each parameter (after the primary parameter).
- Any number of blanks can occur at the beginning or end of the command, and between parameters, punctuation, and values. For example, the following command is valid:

  ```
  ALTER QLOCAL  ('Account'  )           TRIGDPTH  (  1)
  ```

  Blanks within a pair of quotation marks are significant.
- Additional commas can appear anywhere where blanks are allowed and are treated as if they were blanks (unless, of course, they are inside strings enclosed by quotation marks).
- Repeated parameters are not allowed. Repeating a parameter with its "NO" version, as in REPLACE NOREPLACE, is also not allowed.
- Strings that contain blanks, lowercase characters or special characters other than:
  - Period (.)
  - Forward slash (/)
  - Underscore (_)
  - Percent sign (%)

  must be enclosed in single quotation marks, unless they are:
  - Generic values ending with an asterisk
  - A single asterisk (for example, TRACE(*))
  - A range specification containing a colon (for example, CLASS(01:03))

  If the string itself contains a single quotation mark, the single quotation mark is represented by two single quotation marks. Lowercase characters not contained within quotation marks are folded to uppercase.
- On platforms other than z/OS, a string containing no characters (that is, two single quotation marks with no space in between) is interpreted as a blank space enclosed in single quotation marks, that is, interpreted in the same way as (' '). The exception to this is if the attribute being used is one of the following:
  - `TOPICSTR`
  - SUB

  - USERDATA
  - SELECTOR

  then two single quotation marks with no space are interpreted as a zero-length string.
- In v7.0, any trailing blanks in those string attributes which are based on MQCHARV types, such as SELECTOR, sub user data, are treated as significant which means that 'abc ' does not equal 'abc'.
- A left parenthesis followed by a right parenthesis, with no significant information in between, for example

  ```
  NAME ( )
  ```

  is not valid except where specifically noted.
- Keywords are not case sensitive: AltER, alter, and ALTER are all acceptable. Anything that is not contained within quotation marks is folded to uppercase.
- Synonyms are defined for some parameters. For example, DEF is always a synonym for DEFINE, so DEF QLOCAL is valid. Synonyms are not, however, just minimum strings; DEFI is not a valid synonym for DEFINE.

  Note: There is no synonym for the DELETE parameter. This is to avoid accidental deletion of objects when using DEF, the synonym for DEFINE.

For an overview of using MQSC commands for administering IBM WebSphere MQ, see "Performing local administration tasks using MQSC commands" on page 71.

MQSC commands use certain special characters to have certain meanings. For more information about these special characters and how to use them, see Characters with special meanings.

To find out how you can build scripts using MQSC commands, see Building command scripts.

For the full list of MQSC commands, see The MQSC commands.

Related tasks
Building command scripts

WebSphere MQ object names

How to use object names in MQSC commands.

In examples, we use some long names for objects. This is to help you identify the type of object you are dealing with.

When you issue MQSC commands, you need specify only the local name of the queue. In our examples, we use queue names such as:

```
ORANGE.LOCAL.QUEUE
```

The `LOCAL.QUEUE` part of the name is to illustrate that this queue is a local queue. It is ***not*** required for the names of local queues in general.

We also use the name `saturn.queue.manager` as a queue manager name. The `queue.manager` part of the name is to illustrate that this object is a queue manager. It is *not* required for the names of queue managers in general.

Case-sensitivity in MQSC commands

MQSC commands, including their attributes, can be written in uppercase or lowercase. Object names in MQSC commands are folded to uppercase (that is, QUEUE and queue are not differentiated), unless the names are enclosed within single quotation marks. If quotation marks are not used, the object is processed with a name in uppercase. See theMQSC reference for more information.

The `runmqsc` command invocation, in common with all WebSphere MQ control commands, is case sensitive in some WebSphere MQ environments. See Using control commands for more information.

Standard input and output

The *standard input device*, also referred to as `stdin`, is the device from which input to the system is taken. Typically this is the keyboard, but you can specify that input is to come from a serial port or a disk file, for example. The *standard output device*, also referred to as `stdout`, is the device to which output from the system is sent. Typically this is a display, but you can redirect output to a serial port or a file.

On operating-system commands and WebSphere MQ control commands, the < operator redirects input. If this operator is followed by a file name, input is taken from the file. Similarly, the > operator redirects output; if this operator is followed by a file name, output is directed to that file.

Using MQSC commands interactively

You can use MQSC commands interactively by using a command window or shell.

To use MQSC commands interactively, open a command window or shell and enter:

```
runmqsc
```

In this command, a queue manager name has not been specified, so the MQSC commands are processed by the default queue manager. If you want to use a different queue manager, specify the queue manager name on the **runmqsc** command. For example, to run MQSC commands on queue manager `jupiter.queue.manager`, use the command:

```
runmqsc jupiter.queue.manager
```

After this, all the MQSC commands you type in are processed by this queue manager, assuming that it is on the same node and is already running.

Now you can type in any MQSC commands, as required. For example, try this one:

```
DEFINE QLOCAL (ORANGE.LOCAL.QUEUE)
```

For commands that have too many parameters to fit on one line, use continuation characters to indicate that a command is continued on the following line:

- A minus sign (-) indicates that the command is to be continued from the start of the following line.
- A plus sign (+) indicates that the command is to be continued from the first nonblank character on the following line.

Command input terminates with the final character of a nonblank line that is not a continuation character. You can also terminate command input explicitly by entering a semicolon (;). (This is especially useful if you accidentally enter a continuation character at the end of the final line of command input.)

Feedback from MQSC commands

When you issue MQSC commands, the queue manager returns operator messages that confirm your actions or tell you about the errors you have made. For example:

```
AMQ8006: WebSphere MQ queue created.
```

This message confirms that a queue has been created.

```
AMQ8405: Syntax error detected at or near end of command segment below:-

AMQ8426: Valid MQSC commands are:
     ALTER
     CLEAR
     DEFINE
     DELETE
     DISPLAY
     END
     PING
     REFRESH
     RESET
     RESOLVE
     RESUME
     START
     STOP
     SUSPEND
      4 : end
```

This message indicates that you have made a syntax error.

These messages are sent to the standard output device. If you have not entered the command correctly, refer to the MQSC reference for the correct syntax.

Ending interactive input of MQSC commands

To stop working with MQSC commands, enter the END command.

Alternatively, you can use the EOF character for your operating system.

Running MQSC commands from text files

Running MQSC commands interactively is suitable for quick tests, but if you have very long commands, or are using a particular sequence of commands repeatedly, consider redirecting `stdin` from a text file.

"Standard input and output" on page 74 contains information about `stdin` and `stdout`. To redirect `stdin` from a text file, first create a text file containing the MQSC commands using your usual text editor. When you use the `runmqsc` command, use the redirection operators. For example, the following command runs a sequence of commands contained in the text file `myprog.in`:

```
runmqsc < myprog.in
```

Similarly, you can also redirect the output to a file. A file containing the MQSC commands for input is called an *MQSC command file*. The output file containing replies from the queue manager is called the *output file*.

To redirect both `stdin` and `stdout` on the `runmqsc` command, use this form of the command:

```
runmqsc < myprog.in > myprog.out
```

This command invokes the MQSC commands contained in the MQSC command file `myprog.in.` Because we have not specified a queue manager name, the MQSC commands run against the default queue manager. The output is sent to the text file `myprog.out`. Figure 12 on page 76 shows an extract from the MQSC command file `myprog.in` and Figure 13 on page 77 shows the corresponding extract of the output in `myprog.out`.

To redirect `stdin` and `stdout` on the `runmqsc` command, for a queue manager (`saturn.queue.manager`) that is not the default, use this form of the command:

```
runmqsc saturn.queue.manager < myprog.in > myprog.out
```

MQSC command files

MQSC commands are written in human-readable form, that is, in ASCII text. Figure 12 on page 76 is an extract from an MQSC command file showing an MQSC command (DEFINE QLOCAL) with its attributes. The MQSC reference contains a description of each MQSC command and its syntax.

```
  .
  .
  .
DEFINE QLOCAL(ORANGE.LOCAL.QUEUE) REPLACE  +
       DESCR(' ') +
       PUT(ENABLED) +
       DEFPRTY(0) +
       DEFPSIST(NO) +
       GET(ENABLED) +
       MAXDEPTH(5000) +
       MAXMSGL(1024) +
       DEFSOPT(SHARED) +
       NOHARDENBO +
       USAGE(NORMAL) +
       NOTRIGGER;
  .
  .
  .
```

Figure 12. Extract from an MQSC command file

For portability among WebSphere MQ environments, limit the line length in MQSC command files to 72 characters. The plus sign indicates that the command is continued on the next line.

MQSC command reports

The **runmqsc** command returns a *report*, which is sent to `stdout`. The report contains:

- A header identifying MQSC commands as the source of the report:

  ```
  Starting MQSC for queue manager jupiter.queue.manager.
  ```

  Where `jupiter.queue.manager` is the name of the queue manager.
- An optional numbered listing of the MQSC commands issued. By default, the text of the input is echoed to the output. Within this output, each command is prefixed by a sequence number, as shown in Figure 13 on page 77. However, you can use the -e flag on the `runmqsc` command to suppress the output.
- A syntax error message for any commands found to be in error.
- An *operator message* indicating the outcome of running each command. For example, the operator message for the successful completion of a DEFINE QLOCAL command is:

  ```
  AMQ8006: WebSphere MQ queue created.
  ```
- Other messages resulting from general errors when running the script file.
- A brief statistical summary of the report indicating the number of commands read, the number of commands with syntax errors, and the number of commands that could not be processed.

  Note: The queue manager attempts to process only those commands that have no syntax errors.

```
Starting MQSC for queue manager jupiter.queue.manager.
 .
 .
     12:     DEFINE QLOCAL('ORANGE.LOCAL.QUEUE') REPLACE  +
       :            DESCR(' ') +
       :            PUT(ENABLED) +
       :            DEFPRTY(0) +
       :            DEFPSIST(NO) +
       :            GET(ENABLED) +
       :            MAXDEPTH(5000) +
       :            MAXMSGL(1024) +
       :            DEFSOPT(SHARED) +
       :            NOHARDENBO +
       :            USAGE(NORMAL) +
       :            NOTRIGGER;
AMQ8006: WebSphere MQ queue created.
       :
 .
 .
```

Figure 13. Extract from an MQSC command report file

Running the supplied MQSC command files

The following MQSC command files are supplied with WebSphere MQ:

amqscos0.tst
: Definitions of objects used by sample programs.

amqscic0.tst
: Definitions of queues for CICS® transactions.

In WebSphere MQ for Windows, these files are located in the directory *MQ_INSTALLATION_PATH*`\tools\mqsc\samples`. *MQ_INSTALLATION_PATH* represents the high-level directory in which WebSphere MQ is installed.

On UNIX and Linux systems these files are located in the directory *MQ_INSTALLATION_PATH*`/samp`. *MQ_INSTALLATION_PATH* represents the high-level directory in which WebSphere MQ is installed.

The command that runs them is:

```
runmqsc < amqscos0.tst >test.out
```

Using runmqsc to verify commands

You can use the `runmqsc` command to verify MQSC commands on a local queue manager without actually running them. To do this, set the -v flag in the `runmqsc` command, for example:

```
runmqsc -v < myprog.in > myprog.out
```

When you invoke `runmqsc` against an MQSC command file, the queue manager verifies each command and returns a report without actually running the MQSC commands. This allows you to check the syntax of the commands in your command file. This is particularly important if you are:

- Running a large number of commands from a command file.
- Using an MQSC command file many times over.

The returned report is similar to that shown in Figure 13 on page 77.

You cannot use this method to verify MQSC commands remotely. For example, if you attempt this command:

```
runmqsc -w 30 -v jupiter.queue.manager < myprog.in > myprog.out
```

the -w flag, which you use to indicate that the queue manager is remote, is ignored, and the command is run locally in verification mode. 30 is the number of seconds that WebSphere MQ waits for replies from the remote queue manager.

Running MQSC commands from batch files

If you have very long commands, or are using a particular sequence of commands repeatedly, consider redirecting `stdin` from a batch file.

To redirect `stdin` from a batch file, first create a batch file containing the MQSC commands using your usual text editor. When you use the `runmqsc` command, use the redirection operators. The following example:

1. Creates a test queue manager, TESTQM
2. Creates a matching CLNTCONN and listener set to use TCP/IP port 1600
3. Creates a test queue, TESTQ
4. Puts a message on the queue, using the amqsputc sample program

```
export MYTEMPQM=TESTQM
export MYPORT=1600
export MQCHLLIB=/var/mqm/qmgrs/$MQTEMPQM/@ipcc

crtmqm $MYTEMPQM
strmqm $MYTEMPQM
runmqlsr -m $MYTEMPQM -t TCP -p $MYPORT &

runmqsc $MYTEMPQM << EOF
  DEFINE CHANNEL(NTLM) CHLTYPE(SVRCONN) TRPTYPE(TCP)
  DEFINE CHANNEL(NTLM) CHLTYPE(CLNTCONN) QMNAME('$MYTEMPQM') CONNAME('hostname($MYPORT)')
  ALTER  CHANNEL(NTLM) CHLTYPE(CLNTCONN)
  DEFINE QLOCAL(TESTQ)
EOF

amqsputc TESTQ $MYTEMPQM << EOF
hello world
EOF

endmqm -i $MYTEMPQM
```

Figure 14. Example script for running MQSC commands from a batch file

Resolving problems with MQSC commands

If you cannot get MQSC commands to run, use the information in this topic to see if any of these common problems apply to you. It is not always obvious what the problem is when you read the error that a command generates.

If you cannot get MQSC commands to run, use the following information to see if any of these common problems apply to you. It is not always obvious what the problem is when you read the error generated.

When you use the `runmqsc` command, remember the following:

- Use the **<** operator to redirect input from a file. If you omit this operator, the queue manager interprets the file name as a queue manager name, and issues the following error message:

  ```
  AMQ8118: WebSphere MQ queue manager does not exist.
  ```

- If you redirect output to a file, use the **>** redirection operator. By default, the file is put in the current working directory at the time `runmqsc` is invoked. Specify a fully-qualified file name to send your output to a specific file and directory.
- Check that you have created the queue manager that is going to run the commands, by using the following command to display all queue managers:

  ```
  dspmq
  ```

- The queue manager must be running. If it is not, start it; (see Starting a queue manager). You get an error message if you try to start a queue manager that is already running.
- Specify a queue manager name on the `runmqsc` command if you have not defined a default queue manager, or you get this error:

  ```
  AMQ8146: WebSphere MQ queue manager not available.
  ```

- You cannot specify an MQSC command as a parameter of the `runmqsc` command. For example, this is not valid:

  ```
  runmqsc DEFINE QLOCAL(FRED)
  ```

- You cannot enter MQSC commands before you issue the `runmqsc` command.
- You cannot run control commands from `runmqsc`. For example, you cannot issue the `strmqm` command to start a queue manager while you are running MQSC commands interactively. If you do this, you receive error messages similar to the following:

  ```
  runmqsc
   .
   .
  Starting MQSC for queue manager jupiter.queue.manager.

      1 : strmqm saturn.queue.manager
  AMQ8405: Syntax error detected at or near end of cmd segment below:-s

  AMQ8426: Valid MQSC commands are:
      ALTER
      CLEAR
      DEFINE
      DELETE
      DISPLAY
      END
      PING
      REFRESH
      RESET
      RESOLVE
      RESUME
      START
      STOP
      SUSPEND
       2 : end
  ```

Working with queue managers

Examples of MQSC commands that you can use to display or alter queue manager attributes.

Displaying queue manager attributes

To display the attributes of the queue manager specified on the **runmqsc** command, use the following MQSC command:

```
DISPLAY QMGR
```

Typical output from this command is shown in Figure 15 on page 80

```
DISPLAY QMGR
     1 : DISPLAY QMGR
AMQ8408: Display Queue Manager details.
   QMNAME(QM1)                              ACCTCONO(DISABLED)
   ACCTINT(1800)                            ACCTMQI(OFF)
   ACCTQ(OFF)                               ACTIVREC(MSG)
   ACTVCONO (DISABLED)                      ACTVTRC (OFF)
   ALTDATE(2012-05-27)                      ALTTIME(16.14.01)
   AUTHOREV(DISABLED)                       CCSID(850)
   CHAD(DISABLED)                           CHADEV(DISABLED)
   CHADEXIT( )                              CHLEV(DISABLED)
   CLWLDATA( )                              CLWLEXIT( )
   CLWLLEN(100)                             CLWLMRUC(999999999)
   CLWLUSEQ(LOCAL)                          CMDEV(DISABLED)
   CMDLEVEL(750)                            COMMANDQ(SYSTEM.ADMIN.COMMAND.QUEUE)
   CONFIGEV(DISABLED)                       CRDATE(2011-05-27)
   CRTIME(16.14.01)                         DEADQ()
   DEFXMITQ( )                              DESCR( )
   DISTL(YES)                               INHIBTEV(DISABLED)
   IPADDRV(IPV4)                            LOCALEV(DISABLED)
   LOGGEREV(DISABLED)                       MARKINT(5000)
   MAXHANDS(256)                            MAXMSGL(4194304)
   MAXPROPL(NOLIMIT)                        MAXPRTY(9)
   MAXUMSGS(10000)                          MONACLS(QMGR)
   MONCHL(OFF)                              MONQ(OFF)
   PARENT( )                                PERFMEV(DISABLED)
   PLATFORM(WINDOWSNT)                         PSRTYCNT(5)
   PSNPMSG(DISCARD)                         PSNPRES(NORMAL)
   PSSYNCPT(IFPER)                          QMID(QM1_2011-05-27_16.14.01)
   PSMODE(ENABLED)                          REMOTEEV(DISABLED)
   REPOS( )                                 REPOSNL( )
   ROUTEREC(MSG)                            SCHINIT(QMGR)
   SCMDSERV(QMGR)                           SSLCRLNL( )
   SSLCRYP( )                               SSLEV(DISABLED)
   SSLFIPS(NO)                              SSLKEYR(C:\Program Files\IBM\WebSphere
MQ\Data\qmgrs\QM1\ssl\key)
   SSLRKEYC(0)                              STATACLS(QMGR)
   STATCHL(OFF)                             STATINT(1800)
   STATMQI(OFF)                             STATQ(OFF)
   STRSTPEV(ENABLED)                        SYNCPT
   TREELIFE(1800)                           TRIGINT(999999999)
```

Figure 15. Typical output from a DISPLAY QMGR command

The ALL parameter is the default on the DISPLAY QMGR command. It displays all the queue manager attributes. In particular, the output tells you the default queue manager name, the dead-letter queue name, and the command queue name.

You can confirm that these queues exist by entering the command:

```
DISPLAY QUEUE (SYSTEM.*)
```

This displays a list of queues that match the stem SYSTEM.*. The parentheses are required.

Altering queue manager attributes

To alter the attributes of the queue manager specified on the **runmqsc** command, use the MQSC command ALTER QMGR, specifying the attributes and values that you want to change. For example, use the following commands to alter the attributes of jupiter.queue.manager:

```
runmqsc jupiter.queue.manager
ALTER QMGR DEADQ (ANOTHERDLQ) INHIBTEV (ENABLED)
```

The ALTER QMGR command changes the dead-letter queue used, and enables inhibit events.

Related reference

Attributes for the queue manager

Working with local queues

This section contains examples of some MQSC commands that you can use to manage local, model, and alias queues.

See the MQSC reference for detailed information about these commands.

Defining a local queue

For an application, the local queue manager is the queue manager to which the application is connected. Queues managed by the local queue manager are said to be local to that queue manager.

Use the MQSC command DEFINE QLOCAL to create a local queue. You can also use the default defined in the default local queue definition, or you can modify the queue characteristics from those of the default local queue.

Note: The default local queue is named SYSTEM.DEFAULT.LOCAL.QUEUE and it was created on system installation.

For example, the DEFINE QLOCAL command that follows defines a queue called ORANGE.LOCAL.QUEUE with these characteristics:

- It is enabled for gets, enabled for puts, and operates on a priority order basis.
- It is an *normal* queue; it is not an initiation queue or transmission queue, and it does not generate trigger messages.
- The maximum queue depth is 5000 messages; the maximum message length is 4194304 bytes.

```
DEFINE QLOCAL (ORANGE.LOCAL.QUEUE) +
       DESCR('Queue for messages from other systems') +
       PUT (ENABLED) +
       GET (ENABLED) +
       NOTRIGGER +
       MSGDLVSQ (PRIORITY) +
       MAXDEPTH (5000) +
       MAXMSGL (4194304) +
       USAGE (NORMAL);
```

Note:

1. With the exception of the value for the description, all the attribute values shown are the default values. We have shown them here for purposes of illustration. You can omit them if you are sure that the defaults are what you want or have not been changed. See also "Displaying default object attributes" on page 82.
2. USAGE (NORMAL) indicates that this queue is not a transmission queue.
3. If you already have a local queue on the same queue manager with the name ORANGE.LOCAL.QUEUE, this command fails. Use the REPLACE attribute if you want to overwrite the existing definition of a queue, but see also "Changing local queue attributes" on page 83.

Defining a dead-letter queue

Each queue manager must have a local queue to be used as a dead-letter queue so that messages that cannot be delivered to their correct destination can be stored for later retrieval. You must tell the queue manager about the dead-letter queue.

To tell the queue manager about the dead-letter queue, specify a dead-letter queue name on the `crtmqm` command (`crtmqm -u DEAD.LETTER.QUEUE`, for example), or by using the DEADQ attribute on the ALTER QMGR command to specify one later. You must define the dead-letter queue before using it.

A sample dead-letter queue called SYSTEM.DEAD.LETTER.QUEUE is available with the product. This queue is automatically created when you create the queue manager. You can modify this definition if required, and rename it.

A dead-letter queue has no special requirements except that:

- It must be a local queue

- Its MAXMSGL (maximum message length) attribute must enable the queue to accommodate the largest messages that the queue manager has to handle **plus** the size of the dead-letter header (MQDLH)

WebSphere MQ provides a dead-letter queue handler that allows you to specify how messages found on a dead-letter queue are to be processed or removed. For further information, see Handling undelivered messages with the WebSphere MQ dead-letter queue handler .

Displaying default object attributes

You can use the DISPLAY QUEUE command to display attributes that were taken from the default object when a WebSphere MQ object was defined.

When you define a WebSphere MQ object, it takes any attributes that you do not specify from the default object. For example, when you define a local queue, the queue inherits any attributes that you omit in the definition from the default local queue, which is called SYSTEM.DEFAULT.LOCAL.QUEUE. To see exactly what these attributes are, use the following command:

```
DISPLAY QUEUE (SYSTEM.DEFAULT.LOCAL.QUEUE)
```

The syntax of this command is different from that of the corresponding DEFINE command. On the DISPLAY command you can give just the queue name, whereas on the DEFINE command you have to specify the type of the queue, that is, QLOCAL, QALIAS, QMODEL, or QREMOTE.

You can selectively display attributes by specifying them individually. For example:

```
DISPLAY QUEUE (ORANGE.LOCAL.QUEUE) +
        MAXDEPTH +
        MAXMSGL +
        CURDEPTH;
```

This command displays the three specified attributes as follows:

```
AMQ8409: Display Queue details.
   QUEUE(ORANGE.LOCAL.QUEUE)               TYPE(QLOCAL)
   CURDEPTH(0)                             MAXDEPTH(5000)
   MAXMSGL(4194304)
```

CURDEPTH is the current queue depth, that is, the number of messages on the queue. This is a useful attribute to display, because by monitoring the queue depth you can ensure that the queue does not become full.

Copying a local queue definition

You can copy a queue definition using the LIKE attribute on the DEFINE command.

For example:

```
DEFINE QLOCAL (MAGENTA.QUEUE) +
       LIKE (ORANGE.LOCAL.QUEUE)
```

This command creates a queue with the same attributes as our original queue ORANGE.LOCAL.QUEUE, rather than those of the system default local queue. Enter the name of the queue to be copied ***exactly*** as it was entered when you created the queue. If the name contains lower case characters, enclose the name in single quotation marks.

You can also use this form of the DEFINE command to copy a queue definition, but substitute one or more changes to the attributes of the original. For example:

```
DEFINE QLOCAL (THIRD.QUEUE) +
       LIKE (ORANGE.LOCAL.QUEUE) +
       MAXMSGL(1024);
```

This command copies the attributes of the queue ORANGE.LOCAL.QUEUE to the queue THIRD.QUEUE, but specifies that the maximum message length on the new queue is to be 1024 bytes, rather than 4194304.

Note:

1. When you use the LIKE attribute on a DEFINE command, you are copying the queue attributes only. You are not copying the messages on the queue.
2. If you a define a local queue, without specifying LIKE, it is the same as DEFINE LIKE(SYSTEM.DEFAULT.LOCAL.QUEUE).

Changing local queue attributes

You can change queue attributes in two ways, using either the ALTER QLOCAL command or the DEFINE QLOCAL command with the REPLACE attribute.

In "Defining a local queue" on page 81, the queue called ORANGE.LOCAL.QUEUE was defined. Suppose, for example, that you want to decrease the maximum message length on this queue to 10,000 bytes.

- Using the ALTER command:

  ```
  ALTER QLOCAL (ORANGE.LOCAL.QUEUE) MAXMSGL(10000)
  ```

  This command changes a single attribute, that of the maximum message length; all the other attributes remain the same.
- Using the DEFINE command with the REPLACE option, for example:

  ```
  DEFINE QLOCAL (ORANGE.LOCAL.QUEUE) MAXMSGL(10000) REPLACE
  ```

  This command changes not only the maximum message length, but also all the other attributes, which are given their default values. The queue is now put enabled whereas previously it was put inhibited. Put enabled is the default, as specified by the queue SYSTEM.DEFAULT.LOCAL.QUEUE.

  If you ***decrease*** the maximum message length on an existing queue, existing messages are not affected. Any new messages, however, must meet the new criteria.

Clearing a local queue

You can use the CLEAR command to clear a local queue.

To delete all the messages from a local queue called MAGENTA.QUEUE, use the following command:

```
CLEAR QLOCAL (MAGENTA.QUEUE)
```

Note: There is no prompt that enables you to change your mind; once you press the Enter key the messages are lost.

You cannot clear a queue if:

- There are uncommitted messages that have been put on the queue under sync point.
- An application currently has the queue open.

Deleting a local queue

You can use the MQSC command DELETE QLOCAL to delete a local queue.

A queue cannot be deleted if it has uncommitted messages on it. However, if the queue has one or more committed messages and no uncommitted messages, it can be deleted only if you specify the PURGE option. For example:

```
DELETE QLOCAL (PINK.QUEUE) PURGE
```

Specifying NOPURGE instead of PURGE ensures that the queue is not deleted if it contains any committed messages.

Browsing queues

WebSphere MQ provides a sample queue browser that you can use to look at the contents of the messages on a queue. The browser is supplied in both source and executable formats.

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

In WebSphere MQ for Windows, the file names and paths for the sample queue browser are as follows:

Source
: *MQ_INSTALLATION_PATH*`\tools\c\samples\`

Executable
: *MQ_INSTALLATION_PATH*`\tools\c\samples\bin\amqsbcg.exe`

In WebSphere MQ for UNIX and Linux, the file names and paths are as follows:

Source
: *MQ_INSTALLATION_PATH*`/samp/amqsbcg0.c`

Executable
: *MQ_INSTALLATION_PATH*`/samp/bin/amqsbcg`

The sample requires two input parameters, the queue name and the queue manager name. For example:

```
amqsbcg SYSTEM.ADMIN.QMGREVENT.tpp01 saturn.queue.manager
```

Typical results from this command are shown in Figure 16 on page 85.

```
AMQSBCG0 - starts here
**********************

 MQOPEN - 'SYSTEM.ADMIN.QMGR.EVENT'


 MQGET of message number 1
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 850
  Format : 'MQEVENT '
  Priority : 0  Persistence : 0
  MsgId : X'414D512073617475726E2E71756575650005D30033563DB8'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'saturn.queue.manager                            '
  ** Identity Context
  UserIdentifier : '            '
  AccountingToken :
   X'0000000000000000000000000000000000000000000000000000000000000000'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '7'
  PutApplName    : 'saturn.queue.manager        '
  PutDate  : '19970417'    PutTime  : '15115208'
  ApplOriginData : '    '

  GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '104'

****   Message      ****

 length - 104 bytes

00000000:  0700 0000 2400 0000 0100 0000 2C00 0000 '....→.......,...'
00000010:  0100 0000 0100 0000 0100 0000 AE08 0000 '................'
00000020:  0100 0000 0400 0000 4400 0000 DF07 0000 '........D.......'
00000030:  0000 0000 3000 0000 7361 7475 726E 2E71 '....0...saturn.q'
00000040:  7565 7565 2E6D 616E 6167 6572 2020 2020 'ueue.manager    '
00000050:  2020 2020 2020 2020 2020 2020 2020 2020 '                '
00000060:  2020 2020 2020 2020                     '                '

 No more messages
 MQCLOSE
 MQDISC
```

Figure 16. Typical results from queue browser

Enabling large queues

IBM WebSphere MQ supports queues larger than 2 GB.

On Windows systems, support for large files is available without any additional enablement. On AIX, HP-UX, Linux, and Solaris systems, you need to explicitly enable large file support before you can create queue files larger than 2 GB. See your operating system documentation for information on how to do this.

Some utilities, such as tar, cannot cope with files greater than 2 GB. Before enabling large file support, check your operating system documentation for information on restrictions on utilities you use.

For information about planning the amount of storage you need for queues, visit the IBM WebSphere MQ website for platform-specific performance reports:

```
https://www.ibm.com/software/integration/ts/mqseries/
```

Working with alias queues

You can define an alias queue to refer indirectly to another queue or topic.

V 7.5.0.8

⚠ **Attention:** Distribution lists do not support the use of alias queues that point to topic objects. From Version 7.5.0, Fix Pack 8, if an alias queue points to a topic object in a distribution list, IBM WebSphere MQ returns MQRC_ALIAS_BASE_Q_TYPE_ERROR.

The queue to which an alias queue refers can be any of the following:

- A local queue (see "Defining a local queue" on page 81).
- A local definition of a remote queue (see "Creating a local definition of a remote queue" on page 110).
- A topic.

An alias queue is not a real queue, but a definition that resolves to a real (or target) queue at run time. The alias queue definition specifies the target queue. When an application makes an MQOPEN call to an alias queue, the queue manager resolves the alias to the target queue name.

An alias queue cannot resolve to another locally defined alias queue. However, an alias queue can resolve to alias queues that are defined elsewhere in clusters of which the local queue manager is a member. See Name resolution for further information.

Alias queues are useful for:

- Giving different applications different levels of access authorities to the target queue.
- Allowing different applications to work with the same queue in different ways. (Perhaps you want to assign different default priorities or different default persistence values.)
- Simplifying maintenance, migration, and workload balancing. (Perhaps you want to change the target queue name without having to change your application, which continues to use the alias.)

For example, assume that an application has been developed to put messages on a queue called MY.ALIAS.QUEUE. It specifies the name of this queue when it makes an MQOPEN request and, indirectly, if it puts a message on this queue. The application is not aware that the queue is an alias queue. For each MQI call using this alias, the queue manager resolves the real queue name, which could be either a local queue or a remote queue defined at this queue manager.

By changing the value of the TARGET attribute, you can redirect MQI calls to another queue, possibly on another queue manager. This is useful for maintenance, migration, and load-balancing.

Defining an alias queue

The following command creates an alias queue:

```
DEFINE QALIAS (MY.ALIAS.QUEUE) TARGET (YELLOW.QUEUE)
```

This command redirects MQI calls that specify MY.ALIAS.QUEUE to the queue YELLOW.QUEUE. The command does not create the target queue; the MQI calls fail if the queue YELLOW.QUEUE does not exist at run time.

If you change the alias definition, you can redirect the MQI calls to another queue. For example:

```
ALTER QALIAS (MY.ALIAS.QUEUE) TARGET (MAGENTA.QUEUE)
```

This command redirects MQI calls to another queue, MAGENTA.QUEUE.

You can also use alias queues to make a single queue (the target queue) appear to have different attributes for different applications. You do this by defining two aliases, one for each application. Suppose there are two applications:

- Application ALPHA can put messages on YELLOW.QUEUE, but is not allowed to get messages from it.
- Application BETA can get messages from YELLOW.QUEUE, but is not allowed to put messages on it.

The following command defines an alias that is put enabled and get disabled for application ALPHA:

```
DEFINE QALIAS (ALPHAS.ALIAS.QUEUE) +
       TARGET (YELLOW.QUEUE) +
       PUT (ENABLED) +
       GET (DISABLED)
```

The following command defines an alias that is put disabled and get enabled for application BETA:

```
DEFINE QALIAS (BETAS.ALIAS.QUEUE) +
       TARGET (YELLOW.QUEUE) +
       PUT (DISABLED) +
       GET (ENABLED)
```

ALPHA uses the queue name ALPHAS.ALIAS.QUEUE in its MQI calls; BETA uses the queue name BETAS.ALIAS.QUEUE. They both access the same queue, but in different ways.

You can use the LIKE and REPLACE attributes when you define queue aliases, in the same way that you use these attributes with local queues.

Using other commands with alias queues

You can use the appropriate MQSC commands to display or alter alias queue attributes, or to delete the alias queue object. For example:

Use the following command to display the alias queue's attributes:

```
DISPLAY QALIAS (ALPHAS.ALIAS.QUEUE)
```

Use the following command to alter the base queue name, to which the alias resolves, where the `force` option forces the change even if the queue is open:

```
ALTER QALIAS (ALPHAS.ALIAS.QUEUE) TARGET(ORANGE.LOCAL.QUEUE) FORCE
```

Use the following command to delete this queue alias:

```
DELETE QALIAS (ALPHAS.ALIAS.QUEUE)
```

You cannot delete an alias queue if an application currently has the queue open. See the MQSC reference for more information about this and other alias queue commands.

Working with model queues

A queue manager creates a *dynamic queue* if it receives an MQI call from an application specifying a queue name that has been defined as a model queue. The name of the new dynamic queue is generated by the queue manager when the queue is created. A *model queue* is a template that specifies the attributes of any dynamic queues created from it. Model queues provide a convenient method for applications to create queues as required.

Defining a model queue

You define a model queue with a set of attributes in the same way that you define a local queue. Model queues and local queues have the same set of attributes, except that on model queues you can specify whether the dynamic queues created are temporary or permanent. (Permanent queues are maintained across queue manager restarts, temporary ones are not.) For example:

```
DEFINE QMODEL (GREEN.MODEL.QUEUE) +
       DESCR('Queue for messages from application X') +
       PUT (DISABLED) +
       GET (ENABLED) +
       NOTRIGGER +
       MSGDLVSQ (FIFO) +
       MAXDEPTH (1000) +
       MAXMSGL (2000) +
       USAGE (NORMAL) +
       DEFTYPE (PERMDYN)
```

This command creates a model queue definition. From the DEFTYPE attribute, you can see that the actual queues created from this template are permanent dynamic queues. Any attributes not specified are automatically copied from the SYSYTEM.DEFAULT.MODEL.QUEUE default queue.

You can use the LIKE and REPLACE attributes when you define model queues, in the same way that you use them with local queues.

Using other commands with model queues

You can use the appropriate MQSC commands to display or alter a model queue's attributes, or to delete the model queue object. For example:

Use the following command to display the model queue's attributes:

```
DISPLAY QUEUE (GREEN.MODEL.QUEUE)
```

Use the following command to alter the model to enable puts on any dynamic queue created from this model:

```
ALTER QMODEL (BLUE.MODEL.QUEUE) PUT(ENABLED)
```

Use the following command to delete this model queue:

```
DELETE QMODEL (RED.MODEL.QUEUE)
```

Working with administrative topics

Use MQSC commands to manage administrative topics.

See MQSC reference for detailed information about these commands.

Related concepts

Administrative topic objects

“Defining an administrative topic” on page 89
Use the MQSC command **DEFINE TOPIC** to create an administrative topic. When defining an administrative topic you can optionally set each topic attribute.

“Displaying administrative topic object attributes” on page 89
Use the MQSC command **DISPLAY TOPIC** to display an administrative topic object.

“Changing administrative topic attributes” on page 90

You can change topic attributes in two ways, using either the **ALTER TOPIC** command or the **DEFINE TOPIC** command with the **REPLACE** attribute.

"Copying an administrative topic definition" on page 90
You can copy a topic definition using the LIKE attribute on the **DEFINE** command.

"Deleting an administrative topic definition" on page 91
You can use the MQSC command **DELETE TOPIC** to delete an administrative topic.

Defining an administrative topic

Use the MQSC command **DEFINE TOPIC** to create an administrative topic. When defining an administrative topic you can optionally set each topic attribute.

Any attribute of the topic that is not explicitly set is inherited from the default administrative topic, SYSTEM.DEFAULT.TOPIC, that was created when the system installation was installed.

For example, the **DEFINE TOPIC** command that follows, defines a topic called **ORANGE.TOPIC** with these characteristics:

- Resolves to the topic string ORANGE. For information about how topic strings can be used, see Combining topic strings.
- Any attribute that is set to ASPARENT uses the attribute as defined by the parent topic of this topic. This action is repeated up the topic tree as far as the root topic, SYSTEM.BASE.TOPIC is found. For more information about topic trees, see Topic trees.

```
DEFINE TOPIC (ORANGE.TOPIC) +
       TOPICSTR (ORANGE) +
       DEFPRTY(ASPARENT) +
       NPMSGDLV(ASPARENT)
```

Note:

- Except for the value of the topic string, all the attribute values shown are the default values. They are shown here only as an illustration. You can omit them if you are sure that the defaults are what you want or have not been changed. See also "Displaying administrative topic object attributes" on page 89.
- If you already have an administrative topic on the same queue manager with the name ORANGE.TOPIC, this command fails. Use the REPLACE attribute if you want to overwrite the existing definition of a topic, but see also "Changing administrative topic attributes" on page 90

Displaying administrative topic object attributes

Use the MQSC command **DISPLAY TOPIC** to display an administrative topic object.

To display all topics, use:

```
DISPLAY TOPIC(ORANGE.TOPIC)
```

You can selectively display attributes by specifying them individually. For example:

```
DISPLAY TOPIC(ORANGE.TOPIC) +
        TOPICSTR +
        DEFPRTY +
        NPMSGDLV
```

This command displays the three specified attributes as follows:

```
AMQ8633: Display topic details.
   TOPIC(ORANGE.TOPIC)                      TYPE(LOCAL)
   TOPICSTR(ORANGE)                         DEFPRTY(ASPARENT)
   NPMSGDLV(ASPARENT)
```

To display the topic ASPARENT values as they are used at Runtime use DISPLAY TPSTATUS. For example, use:

```
DISPLAY TPSTATUS(ORANGE) DEFPRTY NPMSGDLV
```

The command displays the following details:

```
AMQ8754: Display topic status details.
   TOPICSTR(ORANGE)                        DEFPRTY(0)
   NPMSGDLV(ALLAVAIL)
```

When you define an administrative topic, it takes any attributes that you do not specify explicitly from the default administrative topic, which is called SYSTEM.DEFAULT.TOPIC. To see what these default attributes are, use the following command:

```
DISPLAY TOPIC (SYSTEM.DEFAULT.TOPIC)
```

Changing administrative topic attributes

You can change topic attributes in two ways, using either the **ALTER TOPIC** command or the **DEFINE TOPIC** command with the **REPLACE** attribute.

If, for example, you want to change the default priority of messages delivered to a topic called ORANGE.TOPIC, to be 5, use either of the following commands.

- Using the **ALTER** command:

  ```
  ALTER TOPIC(ORANGE.TOPIC) DEFPRTY(5)
  ```

  This command changes a single attribute, that of the default priority of message delivered to this topic to 5; all other attributes remain the same.
- Using the **DEFINE** command:

  ```
  DEFINE TOPIC(ORANGE.TOPIC) DEFPRTY(5) REPLACE
  ```

  This command changes the default priority of messages delivered to this topic. All the other attributes are given their default values.

If you alter the priority of messages sent to this topic, existing messages are not affected. Any new message, however, use the specified priority if not provided by the publishing application.

Copying an administrative topic definition

You can copy a topic definition using the LIKE attribute on the **DEFINE** command.

For example:

```
DEFINE TOPIC (MAGENTA.TOPIC) +
       LIKE (ORANGE.TOPIC)
```

This command creates a topic, MAGENTA.TOPIC, with the same attributes as the original topic, ORANGE.TOPIC, rather than those of the system default administrative topic. Enter the name of the topic to be copied exactly as it was entered when you created the topic. If the name contains lowercase characters, enclose the name in single quotation marks.

You can also use this form of the **DEFINE** command to copy a topic definition, but make changes to the attributes of the original. For example:

```
DEFINE TOPIC(BLUE.TOPIC) +
       TOPICSTR(BLUE) +
       LIKE(ORANGE.TOPIC)
```

You can also copy the attributes of the topic BLUE.TOPIC to the topic GREEN.TOPIC and specify that when publications cannot be delivered to their correct subscriber queue they are not placed onto the dead-letter queue. For example:

```
DEFINE TOPIC(GREEN.TOPIC) +
       TOPICSTR(GREEN) +
       LIKE(BLUE.TOPIC) +
       USEDLQ(NO)
```

Deleting an administrative topic definition

You can use the MQSC command **DELETE TOPIC** to delete an administrative topic.

```
DELETE TOPIC(ORANGE.TOPIC)
```

Applications will no longer be able to open the topic for publication or make new subscriptions using the object name, ORANGE.TOPIC. Publishing applications that have the topic open are able to continue publishing the resolved topic string. Any subscriptions already made to this topic continue receiving publications after the topic has been deleted.

Applications that are not referencing this topic object but are using the resolved topic string that this topic object represented, 'ORANGE' in this example, continue to work. In this case they inherit the properties from a topic object higher in the topic tree. For more information about topic trees, see Topic trees.

Working with subscriptions

Use MQSC commands to manage subscriptions.

Subscriptions can be one of three types, defined in the **SUBTYPE** attribute:

ADMIN
Administratively defined by a user.

PROXY
An internally created subscription for routing publications between queue managers.

API
Created programmatically, for example, using the MQI MQSUB call.

See the MQSC reference for detailed information about these commands.

Related concepts

"Defining an administrative subscription" on page 92
Use the MQSC command **DEFINE SUB** to create an administrative subscription. You can also use the default defined in the default local subscription definition. Or, you can modify the subscription characteristics from those of the default local subscription, SYSTEM.DEFAULT.SUB that was created when the system was installed.

"Displaying attributes of subscriptions" on page 92
You can use the **DISPLAY SUB** command to display configured attributes of any subscription known to the queue manager.

"Changing local subscription attributes" on page 93
You can change subscription attributes in two ways, using either the **ALTER SUB** command or the **DEFINE SUB** command with the **REPLACE** attribute.

"Copying a local subscription definition" on page 94
You can copy a subscription definition using the **LIKE** attribute on the **DEFINE** command.

"Deleting a subscription" on page 94

You can use the MQSC command **DELETE SUB** to delete a local subscription.

Defining an administrative subscription

Use the MQSC command **DEFINE SUB** to create an administrative subscription. You can also use the default defined in the default local subscription definition. Or, you can modify the subscription characteristics from those of the default local subscription, SYSTEM.DEFAULT.SUB that was created when the system was installed.

For example, the **DEFINE SUB** command that follows defines a subscription called ORANGE with these characteristics:

- Durable subscription, meaning that it persists over queue manager restart, with unlimited expiry.
- Receive publications made on the ORANGE topic string, with the message priorities as set by the publishing applications.
- Publications delivered for this subscription are sent to the local queue SUBQ, this queue must be defined before the definition of the subscription.

```
DEFINE SUB (ORANGE) +
       TOPICSTR (ORANGE) +
       DESTCLAS (PROVIDED) +
       DEST (SUBQ) +
       EXPIRY (UNLIMITED) +
       PUBPRTY (ASPUB)
```

Note:

- The subscription and topic string name do not have to match.
- Except for the values of the description and topic string, all the attribute values shown are the default values. They are shown here only as an illustration. You can omit them if you are sure that the defaults are what you want or have not been changed. See also "Displaying attributes of subscriptions" on page 92.
- If you already have a local subscription on the same queue manager with the name TEST, this command fails. Use the **REPLACE** attribute if you want to overwrite the existing definition of a queue, but see also "Changing local subscription attributes" on page 93.
- If the queue SUBQ does not exist, this command fails.

Displaying attributes of subscriptions

You can use the **DISPLAY SUB** command to display configured attributes of any subscription known to the queue manager.

For example, use:

```
DISPLAY SUB (ORANGE)
```

You can selectively display attributes by specifying them individually. For example:

```
DISPLAY SUB (ORANGE) +
        SUBID +
        TOPICSTR +
        DURABLE
```

This command displays the three specified attributes as follows:

```
AMQ8096: WebSphere MQ subscription inquired.
   SUBID(414D5120414141202020202020202020EE921E4E20002A03)
   SUB(ORANGE)                               TOPICSTR(ORANGE)
   DURABLE(YES)
```

TOPICSTR is the resolved topic string on which this subscriber is operating. When a subscription is defined to use a topic object the topic string from that object is used as a prefix to the topic string provided when making the subscription. SUBID is a unique identifier assigned by the queue manager when a subscription is created. This is a useful attribute to display because some subscription names might be long or in a different character sets for which it might become impractical.

An alternate method for displaying subscriptions is to use the SUBID:

```
DISPLAY SUB +
        SUBID(414D5120414141202020202020202020EE921E4E20002A03) +
        TOPICSTR +
        DURABLE
```

This command gives the same output as before:

```
AMQ8096: WebSphere MQ subscription inquired.
 SUBID(414D5120414141202020202020202020EE921E4E20002A03)
   SUB(ORANGE)                               TOPICSTR(ORANGE)
   DURABLE(YES)
```

Proxy subscriptions on a queue manager are not displayed by default. To display them specify a **SUBTYPE** of PROXY or ALL.

You can use the DISPLAY SBSTATUS command to display the Runtime attributes. For example, use the command:

```
DISPLAY SBSTATUS(ORANGE) NUMMSGS
```

The following output is displayed:

```
AMQ8099: WebSphere MQ subscription status inquired.
   SUB(ORANGE)
   SUBID(414D5120414141202020202020202020EE921E4E20002A03)
   NUMMSGS(0)
```

When you define an administrative subscription, it takes any attributes that you do not specify explicitly from the default subscription, which is called SYSTEM.DEFAULT.SUB. To see what these default attributes are, use the following command:

```
DISPLAY SUB (SYSTEM.DEFAULT.SUB)
```

Changing local subscription attributes

You can change subscription attributes in two ways, using either the **ALTER SUB** command or the **DEFINE SUB** command with the **REPLACE** attribute.

If, for example, you want to change the priority of messages delivered to a subscription called ORANGE to be 5, use either of the following commands:

- Using the ALTER command:

  ```
  ALTER SUB(ORANGE) PUBPRTY(5)
  ```

  This command changes a single attribute, that of the priority of messages delivered to this subscription to 5; all other attributes remain the same.
- Using the DEFINE command:

  ```
  DEFINE SUB (ORANGE) PUBPRTY(5) REPLACE
  ```

  This command changes not only the priority of messages delivered to this subscription, but all the other attributes which are given their default values.

If you alter the priority of messages sent to this subscription, existing messages are not affected. Any new messages, however, are of the specified priority.

Copying a local subscription definition

You can copy a subscription definition using the **LIKE** attribute on the **DEFINE** command.

For example:

```
DEFINE SUB (BLUE) +
       LIKE (ORANGE)
```

You can also copy the attributes of the sub REAL to the sub THIRD.SUB, and specify that the correlID of delivered publications is THIRD, rather than the publishers correlID. For example:

```
DEFINE SUB(THIRD.SUB) +
       LIKE(BLUE) +
       DESTCORL(ORANGE)
```

Deleting a subscription

You can use the MQSC command **DELETE SUB** to delete a local subscription.

```
DELETE SUB(ORANGE)
```

You can also delete a subscription using the SUBID:

```
DELETE SUB SUBID(414D5120414141202020202020202020EE921E4E20002A03)
```

Checking messages on a subscription

About this task

When a subscription is defined it is associated with a queue. Published messages matching this subscription are put to this queue.

Note that the following **runmqsc** commands show only those subscriptions that received messages.

To check for messages currently queued for a subscription perform the following steps:

Procedure

1. To check for messages queued for a subscription type **DISPLAY SBSTATUS(<sub_name>) NUMMSGS**, see “Displaying attributes of subscriptions” on page 92.
2. If the **NUMMSGS** value is greater than zero identify the queue associated with the subscription by typing **DISPLAY SUB(<sub_name>)DEST**.
3. Using the name of the queue returned you can view the messages by following the technique described in “Browsing queues” on page 84.

Working with services

Service objects are a means by which additional processes can be managed as part of a queue manager. With services, you can define programs that are started and stopped when the queue manager starts and ends. IBM WebSphere MQ services are always started under the user ID of the user who started the queue manager.

Service objects can be either of the following types:

Server
A server is a service object that has the parameter SERVTYPE specified as SERVER. A server service object is the definition of a program that is executed when a specified queue manager is started. Server service objects define programs that typically run for a long time. For example, a server service object can be used to execute a trigger monitor process, such as `runmqtrm`.

Only one instance of a server service object can run concurrently. The status of running server service objects can be monitored using the MQSC command, DISPLAY SVSTATUS.

Command
A command is a service object that has the parameter SERVTYPE specified as COMMAND. Command service objects are similar to server service objects, however multiple instances of a command service object can run concurrently, and their status cannot be monitored using the MQSC command DISPLAY SVSTATUS.

If the MQSC command, STOP SERVICE, is executed no check is made to determine whether the program started by the MQSC command, START SERVICE, is still active before executing the stop program.

Defining a service object

You define a service object with various attributes.

The attributes are as follows:

SERVTYPE
Defines the type of the service object. Possible values are as follows:

SERVER
A server service object.

Only one instance of a server service object can be executed at a time. The status of server service objects can be monitored using the MQSC command, DISPLAY SVSTATUS.

COMMAND
A command service object.

Multiple instances of a command service object can be executed concurrently. The status of a command service objects cannot be monitored.

STARTCMD
The program that is executed to start the service. A fully qualified path to the program must be specified.

STARTARG
Arguments passed to the start program.

STDERR
Specifies the path to a file to which the standard error (stderr) of the service program should be redirected.

STDOUT
Specifies the path to a file to which the standard output (stdout) of the service program should be redirected.

STOPCMD
The program that is executed to stop the service. A fully qualified path to the program must be specified.

STOPARG
Arguments passed to the stop program.

CONTROL
Specifies how the service is to be started and stopped:

MANUAL
The service is not to be started automatically or stopped automatically. It is controlled by use of the START SERVICE and STOP SERVICE commands. This is the default value.

QMGR
The service being defined is to be started and stopped at the same time as the queue manager is started and stopped.

STARTONLY
The service is to be started at the same time as the queue manager is started, but is not requested to stop when the queue manager is stopped.

Related concepts

“Managing services” on page 96
By using the CONTROL parameter, an instance of a service object can be either started and stopped automatically by the queue manager, or started and stopped using the MQSC commands START SERVICE and STOP SERVICE.

Managing services

By using the CONTROL parameter, an instance of a service object can be either started and stopped automatically by the queue manager, or started and stopped using the MQSC commands START SERVICE and STOP SERVICE.

When an instance of a service object is started, a message is written to the queue manager error log containing the name of the service object and the process ID of the started process. An example log entry for a server service object starting follows:

```
  02/15/2005 11:54:24 AM - Process(10363.1) User(mqm) Program(amqzmgr0)
Host(HOST_1) Installation(Installation1)
VRMF(7.1.0.0) QMgr(A.B.C)
AMQ5028: The Server 'S1' has started. ProcessId(13031).

  EXPLANATION:
  The Server process has started.
  ACTION:
  None.
```

An example log entry for a command service object starting follows:

```
  02/15/2005 11:53:55 AM - Process(10363.1) User(mqm) Program(amqzmgr0)
Host(HOST_1) Installation(Installation1)
VRMF(7.1.0.0) QMgr(A.B.C)
AMQ5030: The Command 'C1' has started. ProcessId(13030).

  EXPLANATION:
  The Command has started.
  ACTION:
  None.
```

When an instance server service stops, a message is written to the queue manager error logs containing the name of the service and the process ID of the ending process. An example log entry for a server service object stopping follows:

```
  02/15/2005 11:54:54 AM - Process(10363.1) User(mqm) Program(amqzmgr0)
Host(HOST_1) Installation(Installation1)
VRMF(7.1.0.0) QMgr(A.B.C)
AMQ5029: The Server 'S1' has ended. ProcessId(13031).

  EXPLANATION:
  The Server process has ended.
  ACTION:
  None.
```

Related reference

“Additional environment variables” on page 97
When a service is started, the environment in which the service process is started is inherited from the environment of the queue manager. It is possible to define additional environment variables to be

set in the environment of the service process by adding the variables you want to define to one of the `service.env` environment override files.

Additional environment variables

When a service is started, the environment in which the service process is started is inherited from the environment of the queue manager. It is possible to define additional environment variables to be set in the environment of the service process by adding the variables you want to define to one of the `service.env` environment override files.

Note:

There are two possible files to which you can add environment variables:

- The machine scope `service.env` file, which is located in `/var/mqm` on UNIX and Linux systems, or in the data directory selected during installation on Windows systems.
- The queue manager scope `service.env` file, which is located in the queue manager data directory. For example, the location of the environment override file for a queue manager named QMNAME is:
  - On UNIX and Linux systems, `/var/mqm/qmgrs/QMNAME/service.env`
  - On Windows systems, `C:\Program Files\IBM\WebSphere MQ\qmgrs\QMNAME\service.env`

Both files are processed, if available, with definitions in the queue manager scope file taking precedence over those definitions in the machine scope file.

Any environment variable can be specified in `service.env`. For example, if the IBM WebSphere MQ service runs a number of commands, it might be useful to set the PATH user variable in the `service.env` file. The values that you set the variable to can't be environment variables; for example CLASSPATH=*%CLASSPATH%* is incorrect. Similarly, on Linux PATH=*$PATH*`:/opt/mqm/bin` would give unexpected results.

CLASSPATH must be capitalized, and the class path statement can contain only literals. Some services (Telemetry for example) set their own class path. The CLASSPATH defined in `service.env` is added to it.

The format of the variables defined in the file,`service.env` is a list of name and value variable pairs. Each variable must be defined on a new line, and each variable is taken as it is explicitly defined, including white space. An example of the file,`service.env` follows:

```
#********************************************************************#
#*                                                                  *#
#* <N_OCO_COPYRIGHT>                                                *#
#* Licensed Materials - Property of IBM                             *#
#*                                                                  *#
#* 63H9336                                                          *#
#* (C) Copyright IBM Corporation 2005, 2025.                        *#
#*                                                                  *#
#* <NOC_COPYRIGHT>                                                  *#
#*                                                                  *#
#********************************************************************#
#***********************************************************************#
#* Module Name: service.env                                            *#
#* Type       : WebSphere MQ service environment file                  *#
#* Function   : Define additional environment variables to be set      *#
#*              for SERVICE programs.                                  *#
#* Usage      : <VARIABLE>=<VALUE>                                     *#
#*                                                                     *#
#***********************************************************************#
MYLOC=/opt/myloc/bin
MYTMP=/tmp
TRACEDIR=/tmp/trace
MYINITQ=ACCOUNTS.INITIATION.QUEUE
```

Related reference

"Replaceable inserts on service definitions" on page 98
In the definition of a service object, it is possible to substitute tokens. Tokens that are substituted are automatically replaced with their expanded text when the service program is executed. Substitute tokens

can be taken from the following list of common tokens, or from any variables that are defined in the file, service.env.

Replaceable inserts on service definitions

In the definition of a service object, it is possible to substitute tokens. Tokens that are substituted are automatically replaced with their expanded text when the service program is executed. Substitute tokens can be taken from the following list of common tokens, or from any variables that are defined in the file, service.env.

The following are common tokens that can be used to substitute tokens in the definition of a service object:

MQ_INSTALL_PATH
The location where WebSphere MQ is installed.

MQ_DATA_PATH
The location of the WebSphere MQ data directory:

- On UNIX and Linux systems, the WebSphere MQ data directory location is `/var/mqm/`
- On Windows systems, the location of the WebSphere MQ data directory is the data directory selected during the installation of WebSphere MQ

QMNAME
The current queue manager name.

MQ_SERVICE_NAME
The name of the service.

MQ_SERVER_PID
This token can only be used by the STOPARG and STOPCMD arguments.

For server service objects this token is replaced with the process id of the process started by the STARTCMD and STARTARG arguments. Otherwise, this token is replaced with 0.

MQ_Q_MGR_DATA_PATH
The location of the queue manager data directory.

MQ_Q_MGR_DATA_NAME
The transformed name of the queue manager. For more information on name transformation, see Understanding WebSphere MQ file names.

To use replaceable inserts, insert the token within + characters into any of the STARTCMD, STARTARG, STOPCMD, STOPARG, STDOUT or STDERR strings. For examples of this, see “Examples on using service objects” on page 98.

Examples on using service objects

The services in this section are written with UNIX style path separator characters, except where otherwise stated.

Using a server service object

This example shows how to define, use, and alter, a server service object to start a trigger monitor.

1. A server service object is defined, using the following MQSC command:

```
DEFINE SERVICE(S1) +
       CONTROL(QMGR) +
       SERVTYPE(SERVER) +
       STARTCMD('+MQ_INSTALL_PATH+bin/runmqtrm') +
       STARTARG('-m +QMNAME+ -q ACCOUNTS.INITIATION.QUEUE') +
       STOPCMD('+MQ_INSTALL_PATH+bin/amqsstop') +
       STOPARG('-m +QMNAME+ -p +MQ_SERVER_PID+')
```

   Where:

   `+MQ_INSTALL_PATH+` is a token representing the installation directory.

    +QMNAME+ is a token representing the name of the queue manager.

    ACCOUNTS.INITIATION.QUEUE is the initiation queue.

    amqsstop is a sample program provided with WebSphere MQ which requests the queue manager to break all connections for the process id. amqsstop generates PCF commands, therefore the command server must be running.

    +MQ_SERVER_PID+ is a token representing the process id passed to the stop program.

   See "Replaceable inserts on service definitions" on page 98 for a list of the common tokens.

2. An instance of the server service object will execute when the queue manager is next started. However, we will start an instance of the server service object immediately with the following MQSC command:

   ```
   START SERVICE(S1)
   ```

3. The status of the server service process is displayed, using the following MQSC command:

   ```
   DISPLAY SVSTATUS(S1)
   ```

4. This example now shows how to alter the server service object and have the updates picked up by manually restarting the server service process. The server service object is altered so that the initiation queue is specified as JUPITER.INITIATION.QUEUE . The following MQSC command is used:

   ```
   ALTER SERVICE(S1) +
         STARTARG('-m +QMNAME+ -q JUPITER.INITIATION.QUEUE')
   ```

   Note: A running service will not pick up any updates to its service definition until it is restarted.

5. The server service process is restarted so that the alteration is picked up, using the following MQSC commands:

   ```
   STOP SERVICE(S1)
   ```

   Followed by:

   ```
   START SERVICE(S1)
   ```

   The server service process is restarted and picks up the alterations made in "4" on page 99.

   Note: The MQSC command, STOP SERVICE, can only be used if a STOPCMD argument is specified in the service definition.

Using a command service object

This example shows how to define a command service object to start a program that writes entries to the operating system's system log when a queue manager is started or stopped.

1. The command service object is defined, using the following MQSC command:

   ```
   DEFINE SERVICE(S2) +
          CONTROL(QMGR) +
          SERVTYPE(COMMAND) +
          STARTCMD('/usr/bin/logger') +
          STARTARG('Queue manager +QMNAME+ starting') +
          STOPCMD('/usr/bin/logger') +
          STOPARG('Queue manager +QMNAME+ stopping')
   ```

   Where:

    logger is the UNIX and Linux system supplied command to write to the system log.

    +QMNAME+ is a token representing the name of the queue manager.

Using a command service object when a queue manager ends only

This example shows how to define a command service object to start a program that writes entries to the operating system's system log when a queue manager is stopped only.

1. The command service object is defined, using the following MQSC command:

```
DEFINE SERVICE(S3) +
       CONTROL(QMGR) +
       SERVTYPE(COMMAND) +
       STOPCMD('/usr/bin/logger') +
       STOPARG('Queue manager +QMNAME+ stopping')
```

   Where:

   `logger` is a sample program provided with WebSphere MQ that can write entries to the operating system's system log.

   +QMNAME+ is a token representing the name of the queue manager.

More on passing arguments

This example shows how to define a server service object to start a program called `runserv` when a queue manager is started.

This example is written with Windows style path separator characters.

One of the arguments that is to be passed to the starting program is a string containing a space. This argument needs to be passed as a single string. To achieve this, double quotation marks are used as shown in the following command to define the command service object:

1. The server service object is defined, using the following MQSC command:

```
   DEFINE SERVICE(S1) SERVTYPE(SERVER) CONTROL(QMGR) +
     STARTCMD('C:\Program Files\Tools\runserv.exe') +
     STARTARG('-m +QMNAME+ -d "C:\Program Files\Tools\"') +
     STDOUT('C:\Program Files\Tools\+MQ_SERVICE_NAME+.out')


DEFINE SERVICE(S4) +
       CONTROL(QMGR) +
       SERVTYPE(SERVER) +
       STARTCMD('C:\Program Files\Tools\runserv.exe') +
       STARTARG('-m +QMNAME+ -d "C:\Program Files\Tools\"') +
       STDOUT('C:\Program Files\Tools\+MQ_SERVICE_NAME+.out')
```

   Where:

   +QMNAME+ is a token representing the name of the queue manager.

   `"C:\Program Files\Tools\"` is a string containing a space, which will be passed as a single string.

Autostarting a Service

This example shows how to define a server service object that can be used to automatically start the Trigger Monitor when the queue manager starts.

1. The server service object is defined, using the following MQSC command:

```
DEFINE SERVICE(TRIG_MON_START) +
       CONTROL(QMGR) +
       SERVTYPE(SERVER) +
       STARTCMD('runmqtrm') +
       STARTARG('-m +QMNAME+ -q +IQNAME+')
```

   Where:

   +QMNAME+ is a token representing the name of the queue manager.

`+IQNAME+` is an environment variable defined by the user in one of the service.env files representing the name of the initiation queue.

Managing objects for triggering

WebSphere MQ enables you to start an application automatically when certain conditions on a queue are met. For example, you might want to start an application when the number of messages on a queue reaches a specified number. This facility is called *triggering*. You have to define the objects that support triggering.

Triggering described in detail in Starting WebSphere MQ applications using triggers.

Defining an application queue for triggering

An application queue is a local queue that is used by applications for messaging, through the MQI. Triggering requires a number of queue attributes to be defined on the application queue.

Triggering itself is enabled by the *`Trigger`* attribute (TRIGGER in MQSC commands). In this example, a trigger event is to be generated when there are 100 messages of priority 5 or greater on the local queue MOTOR.INSURANCE.QUEUE, as follows:

```
DEFINE QLOCAL (MOTOR.INSURANCE.QUEUE) +
       PROCESS (MOTOR.INSURANCE.QUOTE.PROCESS) +
       MAXMSGL (2000) +
       DEFPSIST (YES) +
       INITQ (MOTOR.INS.INIT.QUEUE) +
       TRIGGER +
       TRIGTYPE (DEPTH) +
       TRIGDPTH (100)+
       TRIGMPRI (5)
```

where:

`QLOCAL (MOTOR.INSURANCE.QUEUE)`
Is the name of the application queue being defined.

`PROCESS (MOTOR.INSURANCE.QUOTE.PROCESS)`
Is the name of the process definition that defines the application to be started by a trigger monitor program.

`MAXMSGL (2000)`
Is the maximum length of messages on the queue.

`DEFPSIST (YES)`
Specifies that messages on this queue are persistent by default.

`INITQ (MOTOR.INS.INIT.QUEUE)`
Is the name of the initiation queue on which the queue manager is to put the trigger message.

`TRIGGER`
Is the trigger attribute value.

`TRIGTYPE (DEPTH)`
Specifies that a trigger event is generated when the number of messages of the required priority (TRIGMPRI) reaches the number specified in TRIGDPTH.

`TRIGDPTH (100)`
Is the number of messages required to generate a trigger event.

`TRIGMPRI (5)`
Is the priority of messages that are to be counted by the queue manager in deciding whether to generate a trigger event. Only messages with priority 5 or higher are counted.

Defining an initiation queue

When a trigger event occurs, the queue manager puts a trigger message on the initiation queue specified in the application queue definition. Initiation queues have no special settings, but you can use the following definition of the local queue MOTOR.INS.INIT.QUEUE for guidance:

```
DEFINE QLOCAL(MOTOR.INS.INIT.QUEUE) +
       GET (ENABLED) +
       NOSHARE +
       NOTRIGGER +
       MAXMSGL (2000) +
       MAXDEPTH (1000)
```

Defining a process

Use the DEFINE PROCESS command to create a process definition. A process definition defines the application to be used to process messages from the application queue. The application queue definition names the process to be used and thereby associates the application queue with the application to be used to process its messages. This is done through the PROCESS attribute on the application queue MOTOR.INSURANCE.QUEUE. The following MQSC command defines the required process, MOTOR.INSURANCE.QUOTE.PROCESS, identified in this example:

```
DEFINE PROCESS (MOTOR.INSURANCE.QUOTE.PROCESS) +
               DESCR ('Insurance request message processing') +
               APPLTYPE (UNIX) +
               APPLICID ('/u/admin/test/IRMP01') +
               USERDATA ('open, close, 235')
```

Where:

`MOTOR.INSURANCE.QUOTE.PROCESS`
: Is the name of the process definition.

`DESCR ('Insurance request message processing')`
: Describes the application program to which this definition relates. This text is displayed when you use the DISPLAY PROCESS command. This can help you to identify what the process does. If you use spaces in the string, you must enclose the string in single quotation marks.

`APPLTYPE (UNIX)`
: Is the type of application to be started.

`APPLICID ('/u/admin/test/IRMP01')`
: Is the name of the application executable file, specified as a fully qualified file name. In Windows systems, a typical APPLICID value would be `c:\appl\test\irmp01.exe`.

`USERDATA ('open, close, 235')`
: Is user-defined data, which can be used by the application.

Displaying attributes of a process definition

Use the DISPLAY PROCESS command to examine the results of your definition. For example:

```
DISPLAY PROCESS (MOTOR.INSURANCE.QUOTE.PROCESS)


    24 : DISPLAY PROCESS (MOTOR.INSURANCE.QUOTE.PROCESS) ALL
AMQ8407: Display Process details.
    DESCR ('Insurance request message processing')
    APPLICID ('/u/admin/test/IRMP01')
    USERDATA (open, close, 235)
    PROCESS (MOTOR.INSURANCE.QUOTE.PROCESS)
    APPLTYPE (UNIX)
```

You can also use the MQSC command ALTER PROCESS to alter an existing process definition, and the DELETE PROCESS command to delete a process definition.

Administering remote IBM WebSphere MQ objects

This section tells you how to administer IBM WebSphere MQ objects on a remote queue manager using MQSC commands, and how to use remote queue objects to control the destination of messages and reply messages.

This section describes:

Channels, clusters, and remote queuing

A queue manager communicates with another queue manager by sending a message and, if required, receiving back a response. The receiving queue manager could be:

- On the same machine
- On another machine in the same location (or even on the other side of the world)
- Running on the same platform as the local queue manager
- Running on another platform supported by WebSphere MQ

These messages might originate from:

- User-written application programs that transfer data from one node to another
- User-written administration applications that use PCF commands or the MQAI
- The IBM WebSphere MQ Explorer.
- Queue managers sending:
  - Instrumentation event messages to another queue manager
  - MQSC commands issued from a `runmqsc` command in indirect mode (where the commands are run on another queue manager)

Before a message can be sent to a remote queue manager, the local queue manager needs a mechanism to detect the arrival of messages and transport them consisting of:

- At least one channel
- A transmission queue
- A channel initiator

For a remote queue manager to received a message, a listener is required.

A channel is a one-way communication link between two queue managers and can carry messages destined for any number of queues at the remote queue manager.

Each end of the channel has a separate definition. For example, if one end is a sender or a server, the other end must be a receiver or a requester. A simple channel consists of a *sender channel definition* at the local queue manager end and a *receiver channel definition* at the remote queue manager end. The two definitions must have the same name and together constitute a single message channel.

If you want the remote queue manager to respond to messages sent by the local queue manager, set up a second channel to send responses back to the local queue manager.

Use the MQSC command DEFINE CHANNEL to define channels. In this section, the examples relating to channels use the default channel attributes unless otherwise specified.

There is a message channel agent (MCA) at each end of a channel, controlling the sending and receiving of messages. The MCA takes messages from the transmission queue and puts them on the communication link between the queue managers.

A transmission queue is a specialized local queue that temporarily holds messages before the MCA picks them up and sends them to the remote queue manager. You specify the name of the transmission queue on a *remote queue definition*.

You can allow an MCA to transfer messages using multiple threads. This process is known as *pipelining*. Pipelining enables the MCA to transfer messages more efficiently, improving channel performance. See Attributes of channels for details of how to configure a channel to use pipelining.

"Preparing channels and transmission queues for remote administration" on page 105 tells you how to use these definitions to set up remote administration.

For more information about setting up distributed queuing in general, see Distributed queuing components.

Remote administration using clusters

In a WebSphere MQ network using distributed queuing, every queue manager is independent. If one queue manager needs to send messages to another queue manager, it must define a transmission queue, a channel to the remote queue manager, and a remote queue definition for every queue to which it wants to send messages.

A *cluster* is a group of queue managers set up in such a way that the queue managers can communicate directly with one another over a single network without complex transmission queue, channel, and queue definitions. Clusters can be set up easily, and typically contain queue managers that are logically related in some way and need to share data or applications. Even the smallest cluster reduces system administration costs.

Establishing a network of queue managers in a cluster involves fewer definitions than establishing a traditional distributed queuing environment. With fewer definitions to make, you can set up or change your network more quickly and easily, and reduce the risk of making an error in your definitions.

To set up a cluster, you need one cluster sender (CLUSSDR) and one cluster receiver (CLUSRCVR) definition for each queue manager. You do not need any transmission queue definitions or remote queue definitions. The principles of remote administration are the same when used within a cluster, but the definitions themselves are greatly simplified.

For more information about clusters, their attributes, and how to set them up, see Queue manager clusters.

Remote administration from a local queue manager

This section tells you how to administer a remote queue manager from a local queue manager using MQSC and PCF commands.

Preparing the queues and channels is essentially the same for both MQSC and PCF commands. In this section, the examples show MQSC commands, because they are easier to understand. For more information about writing administration programs using PCF commands, see "Using Programmable Command Formats" on page 10.

You send MQSC commands to a remote queue manager either interactively or from a text file containing the commands. The remote queue manager might be on the same machine or, more typically, on a different machine. You can remotely administer queue managers in other WebSphere MQ environments, including UNIX and Linux systems, Windows systems, IBM i, and z/OS.

To implement remote administration, you must create specific objects. Unless you have specialized requirements, the default values (for example, for maximum message length) are sufficient.

Preparing queue managers for remote administration

How to use MQSC commands to prepare queue managers for remote administration.

Figure 17 on page 105 shows the configuration of queue managers and channels that you need for remote administration using the **runmqsc** command. The object `source.queue.manager` is the source queue manager from which you can issue MQSC commands and to which the results of these commands (operator messages) are returned. The object `target.queue.manager` is the name of the target queue manager, which processes the commands and generates any operator messages.

Note: If you are using **runmqsc** with the -w option, `source.queue.manager` ***must*** be the default queue manager. For further information on creating a queue manager, see crtmqm.

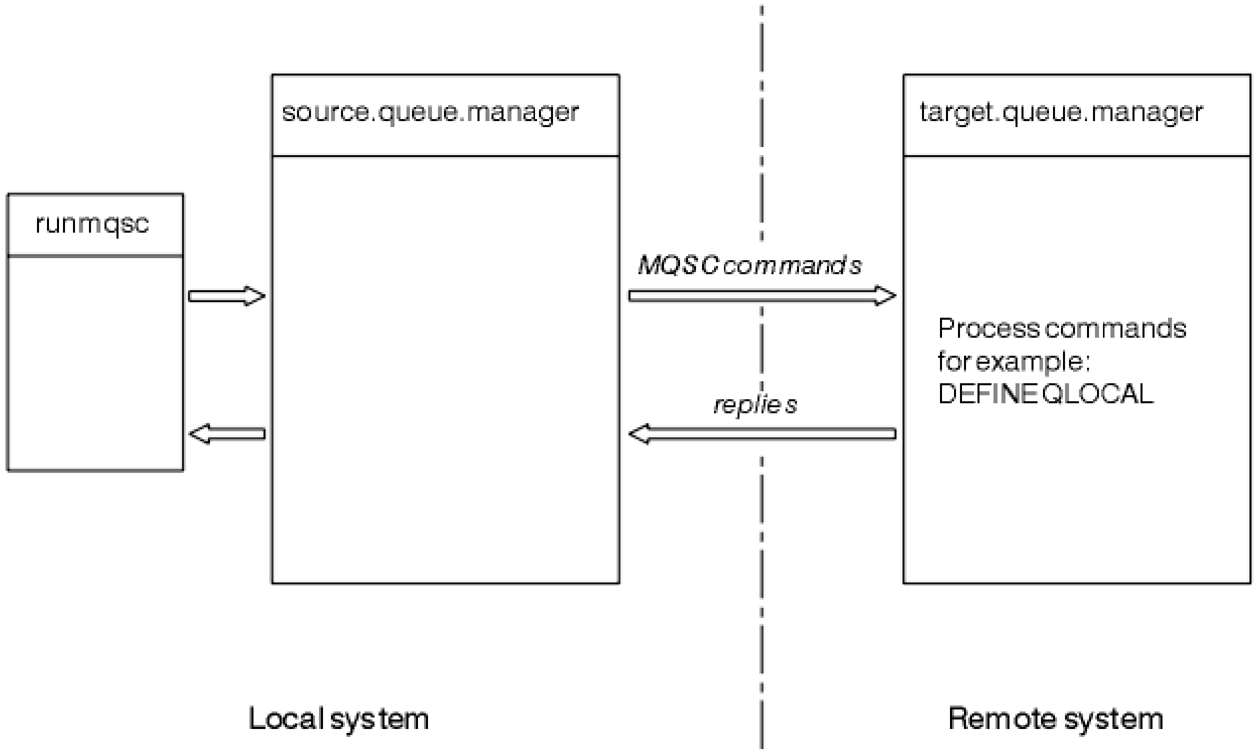


Figure 17. Remote administration using MQSC commands

On both systems, if you have not already done so:

- Create the queue manager and the default objects, using the `crtmqm` command.
- Start the queue manager, using the `strmqm` command.

On the target queue manager:

- The command queue, SYSTEM.ADMIN.COMMAND.QUEUE, must be present. This queue is created by default when a queue manager is created.

You have to run these commands locally or over a network facility such as Telnet.

Preparing channels and transmission queues for remote administration

How to use MQSC commands to prepare channels and transmission queues for remote administration.

To run MQSC commands remotely, set up two channels, one for each direction, and their associated transmission queues. This example assumes that you are using TCP/IP as the transport type and that you know the TCP/IP address involved.

The channel `source.to.target` is for sending MQSC commands from the source queue manager to the target queue manager. Its sender is at `source.queue.manager` and its receiver is at `target.queue.manager`. The channel `target.to.source` is for returning the output from commands and any operator messages that are generated to the source queue manager. You must also define a transmission queue for each channel. This queue is a local queue that is given the name of the receiving queue manager. The XMITQ name must match the remote queue manager name in order for remote administration to work, unless you are using a queue manager alias. Figure 18 on page 106 summarizes this configuration.

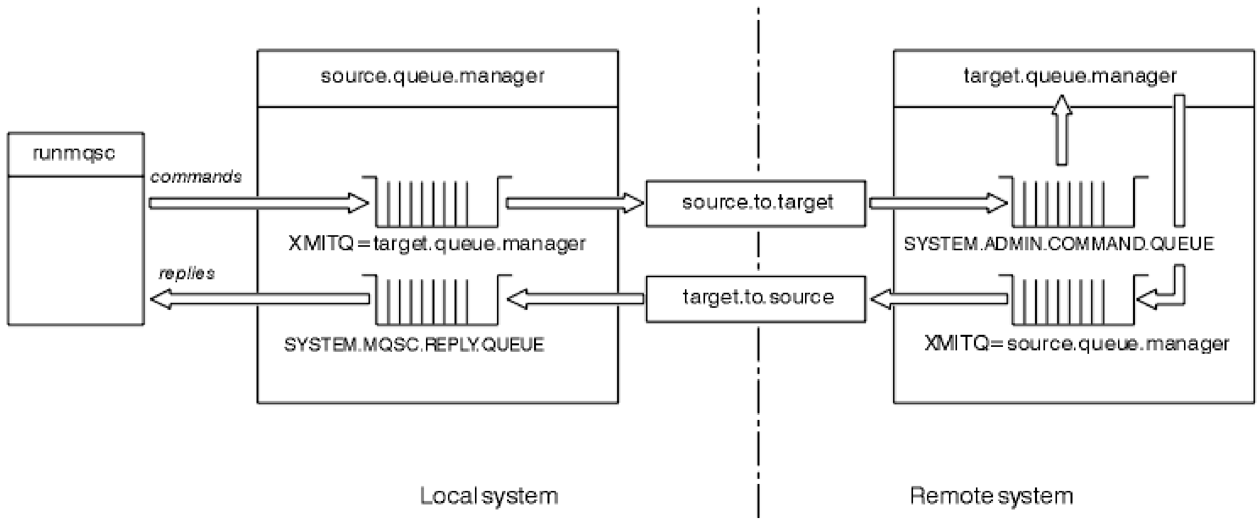


Figure 18. Setting up channels and queues for remote administration

See Connecting applications using distributed queuing for more information about setting up channels.

Defining channels, listeners, and transmission queues

On the source queue manager (`source.queue.manager`), issue the following MQSC commands to define the channels, listener, and the transmission queue:

1. Define the sender channel at the source queue manager:

```
DEFINE CHANNEL ('source.to.target') +
       CHLTYPE(SDR) +
       CONNAME (RHX5498) +
       XMITQ ('target.queue.manager') +
       TRPTYPE(TCP)
```

2. Define the receiver channel at the source queue manager:

```
DEFINE CHANNEL ('target.to.source') +
       CHLTYPE(RCVR) +
       TRPTYPE(TCP)
```

3. Define the listener on the source queue manager:

```
DEFINE LISTENER ('source.queue.manager') +
       TRPTYPE (TCP)
```

4. Define the transmission queue on the source queue manager:

```
DEFINE QLOCAL ('target.queue.manager') +
       USAGE (XMITQ)
```

Issue the following commands on the target queue manager (`target.queue.manager`), to create the channels, listener, and the transmission queue:

1. Define the sender channel on the target queue manager:

```
DEFINE CHANNEL ('target.to.source') +
       CHLTYPE(SDR) +
       CONNAME (RHX7721) +
       XMITQ ('source.queue.manager') +
       TRPTYPE(TCP)
```

2. Define the receiver channel on the target queue manager:

```
DEFINE CHANNEL ('source.to.target') +
       CHLTYPE(RCVR) +
       TRPTYPE(TCP)
```

3. Define the listener on the target queue manager:

```
DEFINE LISTENER ('target.queue.manager') +
       TRPTYPE (TCP)
```

4. Define the transmission queue on the target queue manager:

```
DEFINE QLOCAL ('source.queue.manager') +
       USAGE (XMITQ)
```

Note: The TCP/IP connection names specified for the CONNAME attribute in the sender channel definitions are for illustration only. This is the network name of the machine at the *other* end of the connection. Use the values appropriate for your network.

Starting the listeners and channels

How to use MQSC commands to start listeners and channels.

Start both listeners by using the following MQSC commands:

1. Start the listener on the source queue manager, `source.queue.manager`, by issuing the following MQSC command:

```
START LISTENER ('source.queue.manager')
```

2. Start the listener on the target queue manager, `target.queue.manager`, by issuing the following MQSC command:

```
START LISTENER ('target.queue.manager')
```

Start both sender channels by using the following MQSC commands:

1. Start the sender channel on the source queue manager, `source.queue.manager`, by issuing the following MQSC command:

```
START CHANNEL ('source.to.target')
```

2. Start the sender channel on the target queue manager, `target.queue.manager`, by issuing the following MQSC command:

```
START CHANNEL ('target.to.source')
```

Automatic definition of channels
You enable automatic definition of receiver and server-connection definitions by updating the queue manager object using the MQSC command, ALTER QMGR (or the PCF command Change Queue Manager).

If WebSphere MQ receives an inbound attach request and cannot find an appropriate receiver or server-connection channel, it creates a channel automatically. Automatic definitions are based on two default definitions supplied with WebSphere MQ: SYSTEM.AUTO.RECEIVER and SYSTEM.AUTO.SVRCONN.

For more information about creating channel definitions automatically, see Preparing channels. For information about automatically defining channels for clusters, see Auto-definition of cluster channels.

Managing the command server for remote administration

How to start, stop, and display the status of the command server. A command server is mandatory for all administration involving PCF commands, the MQAI, and also for remote administration.

Each queue manager can have a command server associated with it. A command server processes any incoming commands from remote queue managers, or PCF commands from applications. It presents the commands to the queue manager for processing and returns a completion code or operator message depending on the origin of the command.

Note: For remote administration, ensure that the target queue manager is running. Otherwise, the messages containing commands cannot leave the queue manager from which they are issued. Instead, these messages are queued in the local transmission queue that serves the remote queue manager. Avoid this situation.

There are separate control commands for starting and stopping the command server. Providing the command server is running, users of WebSphere MQ for Windows or WebSphere MQ for Linux (x86 and x86-64 platforms) can perform the operations described in the following sections using the IBM WebSphere MQ Explorer. For more information, see "Administration using the IBM WebSphere MQ Explorer" on page 56.

Starting the command server

Depending on the value of the queue manager attribute, *SCMDSERV*, the command server is either started automatically when the queue manager starts, or must be started manually. The value of the queue manager attribute can be altered using the MQSC command `ALTER QMGR` specifying the parameter `SCMDSERV`. By default, the command server is started automatically.

If *SCMDSERV* is set to MANUAL, start the command server using the command:

```
strmqcsv saturn.queue.manager
```

where `saturn.queue.manager` is the queue manager for which the command server is being started.

Displaying the status of the command server

For remote administration, ensure that the command server on the target queue manager is running. If it is not running, remote commands cannot be processed. Any messages containing commands are queued in the target queue manager's command queue.

To display the status of the command server for a queue manager, issue the following MQSC command:

```
DISPLAY QMSTATUS CMDSERV
```

Stopping a command server

To end the command server started by the previous example use the following command:

```
endmqcsv saturn.queue.manager
```

You can stop the command server in two ways:

- For a controlled stop, use the `endmqcsv` command with the -c flag, which is the default.
- For an immediate stop, use the `endmqcsv` command with the -i flag.

Note: Stopping a queue manager also ends the command server associated with it.

Issuing MQSC commands on a remote queue manager

You can use a particular form of the **runmqsc** command to run MQSC commands on a remote queue manager.

The command server ***must*** be running on the target queue manager, if it is going to process MQSC commands remotely. (This is not necessary on the source queue manager). For information on how to start the command server on a queue manager, see "Managing the command server for remote administration" on page 108.

On the source queue manager, you can then run MQSC commands interactively in indirect mode by typing:

```
runmqsc -w 30 target.queue.manager
```

This form of the `runmqsc` command, with the -w flag, runs the MQSC commands in indirect mode, where commands are put (in a modified form) on the command server input queue and executed in order.

When you type in an MQSC command, it is redirected to the remote queue manager, in this case, `target.queue.manager`. The timeout is set to 30 seconds; if a reply is not received within 30 seconds, the following message is generated on the local (source) queue manager:

```
AMQ8416: MQSC timed out waiting for a response from the command server.
```

When you stop issuing MQSC commands, the local queue manager displays any timed-out responses that have arrived and discards any further responses.

The source queue manager defaults to the default local queue manager. If you specify the `-m` *LocalQmgrName* option in the **runmqsc** command, you can direct the commands to be issued by way of any local queue manager.

In indirect mode, you can also run an MQSC command file on a remote queue manager. For example:

```
runmqsc -w 60 target.queue.manager < mycomds.in > report.out
```

where `mycomds.in` is a file containing MQSC commands and `report.out` is the report file.

Suggested method for issuing commands remotely

When you are issuing commands on a remote queue manager, consider using the following approach:

1. Put the MQSC commands to be run on the remote system in a command file.
2. Verify your MQSC commands locally, by specifying the -v flag on the `runmqsc` command.

   You cannot use `runmqsc` to verify MQSC commands on another queue manager.
3. Check that the command file runs locally without error.
4. Run the command file on the remote system.

If you have problems using MQSC commands remotely

If you have difficulty in running MQSC commands remotely, make sure that you have:

- Started the command server on the target queue manager.
- Defined a valid transmission queue.
- Defined the two ends of the message channels for both:
  - The channel along which the commands are being sent.
  - The channel along which the replies are to be returned.
- Specified the correct connection name (CONNAME) in the channel definition.
- Started the listeners before you started the message channels.
- Checked that the disconnect interval has not expired, for example, if a channel started but then shut down after some time. This is especially important if you start the channels manually.
- Sent requests from a source queue manager that do not make sense to the target queue manager (for example, requests that include parameters that are not supported on the remote queue manager).

See also "Resolving problems with MQSC commands" on page 78.

Creating a local definition of a remote queue

A local definition of a remote queue is a definition on a local queue manager that refers to a queue on a remote queue manager.

You do not have to define a remote queue from a local position, but the advantage of doing so is that applications can refer to the remote queue by its locally-defined name instead of having to specify a name that is qualified by the ID of the queue manager on which the remote queue is located.

Understanding how local definitions of remote queues work

An application connects to a local queue manager and then issues an `MQOPEN` call. In the open call, the queue name specified is that of a remote queue definition on the local queue manager. The remote queue definition supplies the names of the target queue, the target queue manager, and optionally, a transmission queue. To put a message on the remote queue, the application issues an `MQPUT` call, specifying the handle returned from the `MQOPEN` call. The queue manager uses the remote queue name and the remote queue manager name in a transmission header at the start of the message. This information is used to route the message to its correct destination in the network.

As administrator, you can control the destination of the message by altering the remote queue definition.

The following example shows how an application puts a message on a queue owned by a remote queue manager. The application connects to a queue manager, for example, `saturn.queue.manager`. The target queue is owned by another queue manager.

On the `MQOPEN` call, the application specifies these fields:

Field value	Description
ObjectName CYAN.REMOTE.QUEUE	Specifies the local name of the remote queue object. This defines the target queue and the target queue manager.
ObjectType (Queue)	Identifies this object as a queue.
ObjectQmgrName Blank or `saturn.queue.manager`	This field is optional. If blank, the name of the local queue manager is assumed. (This is the queue manager on which the remote queue definition exists.)

After this, the application issues an `MQPUT` call to put a message onto this queue.

On the local queue manager, you can create a local definition of a remote queue using the following MQSC commands:

```
DEFINE QREMOTE (CYAN.REMOTE.QUEUE) +
       DESCR ('Queue for auto insurance requests from the branches') +
       RNAME (AUTOMOBILE.INSURANCE.QUOTE.QUEUE) +
       RQMNAME (jupiter.queue.manager) +
       XMITQ (INQUOTE.XMIT.QUEUE)
```

where:

QREMOTE (CYAN.REMOTE.QUEUE)
Specifies the local name of the remote queue object. This is the name that applications connected to this queue manager must specify in the MQOPEN call to open the queue AUTOMOBILE.INSURANCE.QUOTE.QUEUE on the remote queue manager `jupiter.queue.manager`.

DESCR ('Queue for auto insurance requests from the branches')
Provides additional text that describes the use of the queue.

RNAME (AUTOMOBILE.INSURANCE.QUOTE.QUEUE)
Specifies the name of the target queue on the remote queue manager. This is the real target queue for messages sent by applications that specify the queue name CYAN.REMOTE.QUEUE. The queue AUTOMOBILE.INSURANCE.QUOTE.QUEUE must be defined as a local queue on the remote queue manager.

RQMNAME (jupiter.queue.manager)
Specifies the name of the remote queue manager that owns the target queue AUTOMOBILE.INSURANCE.QUOTE.QUEUE.

XMITQ (INQUOTE.XMIT.QUEUE)
Specifies the name of the transmission queue. This is optional; if the name of a transmission queue is not specified, a queue with the same name as the remote queue manager is used.

In either case, the appropriate transmission queue must be defined as a local queue with a *Usage* attribute specifying that it is a transmission queue (USAGE(XMITQ) in MQSC commands).

An alternative way of putting messages on a remote queue

Using a local definition of a remote queue is not the only way of putting messages on a remote queue. Applications can specify the full queue name, including the remote queue manager name, as part of the MQOPEN call. In this case, you do not need a local definition of a remote queue. However, this means that applications must either know, or have access to, the name of the remote queue manager at run time.

Using other commands with remote queues

You can use MQSC commands to display or alter the attributes of a remote queue object, or you can delete the remote queue object. For example:

- To display the remote queue's attributes:

  ```
  DISPLAY QUEUE (CYAN.REMOTE.QUEUE)
  ```

- To change the remote queue to enable puts. This does not affect the target queue, only applications that specify this remote queue:

  ```
  ALTER QREMOTE (CYAN.REMOTE.QUEUE) PUT(ENABLED)
  ```

- To delete this remote queue. This does not affect the target queue, only its local definition:

  ```
   DELETE QREMOTE (CYAN.REMOTE.QUEUE)
  ```

Note: When you delete a remote queue, you delete only the local representation of the remote queue. You do not delete the remote queue itself or any messages on it.

Defining a transmission queue

A transmission queue is a local queue that is used when a queue manager forwards messages to a remote queue manager through a message channel.

The channel provides a one-way link to the remote queue manager. Messages are queued at the transmission queue until the channel can accept them. When you define a channel, you must specify a transmission queue name at the sending end of the message channel.

The MQSC command attribute USAGE defines whether a queue is a transmission queue or a normal queue.

Default transmission queues

When a queue manager sends messages to a remote queue manager, it identifies the transmission queue using the following sequence:

1. The transmission queue named on the XMITQ attribute of the local definition of a remote queue.
2. A transmission queue with the same name as the target queue manager. (This value is the default value on XMITQ of the local definition of a remote queue.)
3. The transmission queue named on the DEFXMITQ attribute of the local queue manager.

For example, the following MQSC command creates a default transmission queue on `source.queue.manager` for messages going to `target.queue.manager`:

```
DEFINE QLOCAL ('target.queue.manager') +
       DESCR ('Default transmission queue for target qm') +
       USAGE (XMITQ)
```

Applications can put messages directly on a transmission queue, or indirectly through a remote queue definition. See also “Creating a local definition of a remote queue” on page 110.

Using remote queue definitions as aliases

In addition to locating a queue on another queue manager, you can also use a local definition of a remote queue for Queue manager aliases and reply-to queue aliases. Both types of alias are resolved through the local definition of a remote queue. You must set up the appropriate channels for the message to arrive at its destination.

Queue manager aliases

An alias is the process by which the name of the target queue manager, as specified in a message, is modified by a queue manager on the message route. Queue manager aliases are important because you can use them to control the destination of messages within a network of queue managers.

You do this by altering the remote queue definition on the queue manager at the point of control. The sending application is not aware that the queue manager name specified is an alias.

For more information about queue manager aliases, see What are aliases?.

Reply-to queue aliases

Optionally, an application can specify the name of a reply-to queue when it puts a *request message* on a queue.

If the application that processes the message extracts the name of the reply-to queue, it knows where to send the *reply message*, if required.

A reply-to queue alias is the process by which a reply-to queue, as specified in a request message, is altered by a queue manager on the message route. The sending application is not aware that the reply-to queue name specified is an alias.

A reply-to queue alias lets you alter the name of the reply-to queue and optionally its queue manager. This in turn lets you control which route is used for reply messages.

For more information about request messages, reply messages, and reply-to queues, see Types of message and Reply-to queue and queue manager.

For more information about reply-to queue aliases, see Reply-to queue aliases and clusters .

Data conversion between coded character sets

Message data in WebSphere MQ defined formats (also known as *built-in formats*) can be converted by the queue manager from one coded character set to another, provided that both character sets relate to a single language or a group of similar languages.

For example, conversion between coded character sets with identifiers (CCSIDs) 850 and 500 is supported, because both apply to Western European languages.

For EBCDIC newline (NL) character conversions to ASCII, see All queue managers .

Supported conversions are defined in Data conversion.

When a queue manager cannot convert messages in built-in formats

The queue manager cannot automatically convert messages in built-in formats if their CCSIDs represent different national-language groups. For example, conversion between CCSID 850 and CCSID 1025 (which is an EBCDIC coded character set for languages using Cyrillic script) is not supported because many of the characters in one coded character set cannot be represented in the other. If you have a network of queue managers working in different national languages, and data conversion among some of the coded character sets is not supported, you can enable a default conversion. Default data conversion is described in "Default data conversion" on page 113.

File ccsid.tbl

The file ccsid.tbl is used for the following purposes:

- In WebSphere MQ for Windows it records all the supported code sets.
- On AIX and HP-UX platforms, the supported code sets are held internally by the operating system.
- For all other UNIX and Linux platforms, the supported code sets are held in conversion tables provided by WebSphere MQ.
- It specifies any additional code sets. To specify additional code sets, you need to edit ccsid.tbl (guidance on how to do this is provided in the file).
- It specifies any default data conversion.

You can update the information recorded in ccsid.tbl; you might want to do this if, for example, a future release of your operating system supports additional coded character sets.

In WebSphere MQ for Windows, ccsid.tbl is located in directory `C:\Program Files\IBM\WebSphere MQ\conv\table` by default.

In WebSphere MQ for UNIX and Linux systems, ccsid.tbl is located in directory `/var/mqm/conv/table`.

Default data conversion

If you set up channels between two machines on which data conversion is not normally supported, you must enable default data conversion for the channels to work.

To enable default data conversion, edit the ccsid.tbl file to specify a default EBCDIC CCSID and a default ASCII CCSID. Instructions on how to do this are included in the file. You must do this on all machines that will be connected using the channels. Restart the queue manager for the change to take effect.

The default data-conversion process is as follows:

- If conversion between the source and target CCSIDs is not supported, but the CCSIDs of the source and target environments are either both EBCDIC or both ASCII, the character data is passed to the target application without conversion.
- If one CCSID represents an ASCII coded character set, and the other represents an EBCDIC coded character set, WebSphere MQ converts the data using the default data-conversion CCSIDs defined in ccsid.tbl.

Note: Try to restrict the characters being converted to those that have the same code values in the coded character set specified for the message and in the default coded character set. If you use only the set of characters that is valid for WebSphere MQ object names (as defined in Naming IBM WebSphere MQ objects) you will, in general, satisfy this requirement. Exceptions occur with EBCDIC CCSIDs 290, 930, 1279, and 5026 used in Japan, where the lowercase characters have different codes from those used in other EBCDIC CCSIDs.

Converting messages in user-defined formats

The queue manager cannot convert messages in user-defined formats from one coded character set to another. If you need to convert data in a user-defined format, you must supply a data-conversion exit for each such format. Do not use default CCSIDs to convert character data in user-defined formats. For more information about converting data in user-defined formats and about writing data conversion exits, see the Writing data-conversion exits.

Changing the queue manager CCSID

When you have used the CCSID attribute of the ALTER QMGR command to change the CCSID of the queue manager, stop and restart the queue manager to ensure that all running applications, including the command server and channel programs, are stopped and restarted.

This is necessary because any applications that are running when the queue manager CCSID is changed continue to use the existing CCSID.

Administering IBM WebSphere MQ Telemetry

IBM WebSphere MQ Telemetry is administered using IBM WebSphere MQ Explorer or at a command line. Use the explorer to configure telemetry channels, control the telemetry service, and monitor the MQTT clients that are connected to IBM WebSphere MQ. Configure the security of IBM WebSphere MQ Telemetry using JAAS, SSL and the IBM WebSphere MQ object authority manager.

Administering using IBM WebSphere MQ Explorer

Use the explorer to configure telemetry channels, control the telemetry service, and monitor the MQTT clients that are connected to IBM WebSphere MQ. Configure the security of IBM WebSphere MQ Telemetry using JAAS, SSL and the IBM WebSphere MQ object authority manager.

Administering using the command line

IBM WebSphere MQ Telemetry can be completely administered at the command line using the IBM WebSphere MQ MQSC commands.

The IBM WebSphere MQ Telemetry documentation also has sample scripts that demonstrate the basic usage of the MQ Telemetry Transport v3 Client application.

Read and understand the samples in IBM WebSphere MQ Telemetry sample programs in the Developing applications for IBM WebSphere MQ Telemetry section before using them.

Related concepts

WebSphere MQ Telemetry

“Configure distributed queuing to send messages to MQTT clients” on page 118
IBM WebSphere MQ applications can send MQTT v3 clients messages by publishing to subscription created by a client, or by sending a message directly. Whichever method is used, the message is placed on SYSTEM.MQTT.TRANSMIT.QUEUE, and sent to the client by the telemetry (MQXR) service. There are a number of ways to place a message on SYSTEM.MQTT.TRANSMIT.QUEUE.

“MQTT client identification, authorization, and authentication” on page 121

“Telemetry channel authentication using SSL” on page 127

“Publication privacy on telemetry channels ” on page 130

“SSL configuration of MQTT clients and telemetry channels” on page 130

“Telemetry channel JAAS configuration” on page 135
Configure JAAS to authenticate the `Username` sent by the client.

“IBM WebSphere MQ Telemetry daemon for devices concepts” on page 137
The IBM WebSphere MQ Telemetry daemon for devices is an advanced MQTT V3 client application. Use it to store and forward messages from other MQTT clients. It connects to IBM WebSphere MQ like an MQTT client, but you can also connect other MQTT clients to it.

Related tasks

“Configuring a queue manager for telemetry on Linux and AIX” on page 115
Follow these manual steps to configure a queue manager to run IBM WebSphere MQ Telemetry. You can run an automated procedure to set up a simpler configuration using the IBM WebSphere MQ Telemetry support for IBM WebSphere MQ Explorer.

“Configuring a queue manager for telemetry on Windows” on page 117
Follow these manual steps to configure a queue manager to run IBM WebSphere MQ Telemetry. You can run an automated procedure to set up a simpler configuration using the IBM WebSphere MQ Telemetry support for IBM WebSphere MQ Explorer.

Related reference

MQXR properties

Configuring a queue manager for telemetry on Linux and AIX

Follow these manual steps to configure a queue manager to run IBM WebSphere MQ Telemetry. You can run an automated procedure to set up a simpler configuration using the IBM WebSphere MQ Telemetry support for IBM WebSphere MQ Explorer.

Before you begin

1. See Installing IBM WebSphere MQ Telemetry for information on how to install IBM WebSphere MQ, and the IBM WebSphere MQ Telemetry feature.
2. Create and start a queue manager. The queue manager is referred to as *qMgr* in this task.
3. As part of this task you configure the telemetry (MQXR) service. The MQXR property settings are stored in a platform-specific properties file: `mqxr_unix.properties`. You do not normally need to edit the MQXR properties file directly, because almost all settings can be configured through MQSC admin commands or MQ Explorer. If you do decide to edit the file directly, stop the queue manager before you make your changes. See MQXR properties.

About this task

The IBM WebSphere MQ Telemetry support for IBM WebSphere MQ Explorer includes a wizard, and a sample command procedure `sampleMQM`. They set up an initial configuration using the guest user ID; see Verifying the installation of IBM WebSphere MQ Telemetry by using IBM WebSphere MQ Explorer and IBM WebSphere MQ Telemetry sample programs.

Follow the steps in this task to configure IBM WebSphere MQ Telemetry manually using different authorization schemes.

Procedure

1. Open a command window at the telemetry samples directory.

   The telemetry samples directory is `/opt/mqm/mqxr/samples`.
2. Create the telemetry transmission queue.

   ```
   echo "DEFINE QLOCAL('SYSTEM.MQTT.TRANSMIT.QUEUE') USAGE(XMITQ) MAXDEPTH(100000)" | runmqsc
   qMgr
   ```

   When the telemetry (MQXR) service is first started, it creates `SYSTEM.MQTT.TRANSMIT.QUEUE`.

   It is created manually in this task, because `SYSTEM.MQTT.TRANSMIT.QUEUE` must exist before the telemetry (MQXR) service is started, to authorize access to it.
3. Set the default transmission queue

   When the telemetry (MQXR) service is first started, it does not alter the queue manager to make `SYSTEM.MQTT.TRANSMIT.QUEUE` the default transmission queue.

   To make `SYSTEM.MQTT.TRANSMIT.QUEUE` the default transmission queue alter the default transmission queue property. Alter the property using the IBM WebSphere MQ Explorer or with the command in the following example:

   ```
   echo "ALTER QMGR DEFXMITQ('SYSTEM.MQTT.TRANSMIT.QUEUE')" | runmqsc qMgr
   ```

   Altering the default transmission queue might interfere with your existing configuration. The reason for altering the default transmission queue to `SYSTEM.MQTT.TRANSMIT.QUEUE` is to make sending messages directly to MQTT clients easier. Without altering the default transmission queue you must add a remote queue definition for every client that receives IBM WebSphere MQ messages; see "Sending a message to a client directly" on page 120.
4. Follow a procedure in "Authorizing MQTT clients to access WebSphere MQ objects" on page 122 to create one or more user IDs. The user IDs have the authority to publish, subscribe, and send publications to MQTT clients.
5. Install the telemetry (MQXR) service

   ```
   cat /opt/<install_dir>/mqxr/samples/installMQXRService_unix.mqsc | runmqsc qMgr
   ```

   See also the example code in Figure 19 on page 117.
6. Start the service

   ```
   echo "START SERVICE(SYSTEM.MQXR.SERVICE)" | runmqsc qMgr
   ```

   The telemetry (MQXR) service is started automatically when the queue manager is started.

   It is started manually in this task, because the queue manager is already running.
7. Using IBM WebSphere MQ Explorer, configure telemetry channels to accept connections from MQTT clients.

   The telemetry channels must be configured such that their identities are one of the user IDs defined in step 4.

   See also DEFINE CHANNEL (MQTT).
8. Verify the configuration by running the sample client.

   For the sample client to work with your telemetry channel, the channel must authorize the client to publish, subscribe, and receive publications. The sample client connects to the telemetry channel on port 1883 by default. See also IBM WebSphere MQ Telemetry sample programs.

Example

Figure 19 on page 117 shows the **runmqsc** command to create the SYSTEM.MQXR.SERVICE manually on Linux.

```
DEF    SERVICE(SYSTEM.MQXR.SERVICE) +
    CONTROL(QMGR) +
    DESCR('Manages clients using MQXR protocols such as MQTT') +
    SERVTYPE(SERVER) +
    STARTCMD('+MQ_INSTALL_PATH+/mqxr/bin/runMQXRService.sh') +
    STARTARG('-m +QMNAME+ -d "+MQ_Q_MGR_DATA_PATH+" -g "+MQ_DATA_PATH+"') +
    STOPCMD('+MQ_INSTALL_PATH+/mqxr/bin/endMQXRService.sh') +
    STOPARG('-m +QMNAME+') +
    STDOUT('+MQ_Q_MGR_DATA_PATH+/mqxr.stdout') +
    STDERR('+MQ_Q_MGR_DATA_PATH+/mqxr.stderr')
```

Figure 19. installMQXRService_unix.mqsc

Configuring a queue manager for telemetry on Windows

Follow these manual steps to configure a queue manager to run IBM WebSphere MQ Telemetry. You can run an automated procedure to set up a simpler configuration using the IBM WebSphere MQ Telemetry support for IBM WebSphere MQ Explorer.

Before you begin

1. See Installing IBM WebSphere MQ Telemetry for information on how to install IBM WebSphere MQ, and the IBM WebSphere MQ Telemetry feature.
2. Create and start a queue manager. The queue manager is referred to as *qMgr* in this task.
3. As part of this task you configure the telemetry (MQXR) service. The MQXR property settings are stored in a platform-specific properties file: mqxr_win.properties. You do not normally need to edit the MQXR properties file directly, because almost all settings can be configured through MQSC admin commands or MQ Explorer. If you do decide to edit the file directly, stop the queue manager before you make your changes. See MQXR properties.

About this task

The IBM WebSphere MQ Telemetry support for IBM WebSphere MQ Explorer includes a wizard, and a sample command procedure sampleMQM. They set up an initial configuration using the guest user ID; see Verifying the installation of IBM WebSphere MQ Telemetry by using IBM WebSphere MQ Explorer and IBM WebSphere MQ Telemetry sample programs.

Follow the steps in this task to configure IBM WebSphere MQ Telemetry manually using different authorization schemes.

Procedure

1. Open a command window at the telemetry samples directory.

   The telemetry samples directory is *WMQ program installation directory*\mqxr\samples.
2. Create the telemetry transmission queue.

   ```
   echo DEFINE QLOCAL('SYSTEM.MQTT.TRANSMIT.QUEUE') USAGE(XMITQ) MAXDEPTH(100000) | runmqsc qMgr
   ```

   When the telemetry (MQXR) service is first started, it creates SYSTEM.MQTT.TRANSMIT.QUEUE.

   It is created manually in this task, because SYSTEM.MQTT.TRANSMIT.QUEUE must exist before the telemetry (MQXR) service is started, to authorize access to it.
3. Set the default transmission queue

   ```
   echo ALTER QMGR DEFXMITQ('SYSTEM.MQTT.TRANSMIT.QUEUE') | runmqsc qMgr
   ```

   Figure 20. Set default transmission queue

When the telemetry (MQXR) service is first started, it does not alter the queue manager to make SYSTEM.MQTT.TRANSMIT.QUEUE the default transmission queue.

To make SYSTEM.MQTT.TRANSMIT.QUEUE the default transmission queue alter the default transmission queue property. Alter the property using the IBM WebSphere MQ Explorer or with the command in Figure 20 on page 117.

Altering the default transmission queue might interfere with your existing configuration. The reason for altering the default transmission queue to SYSTEM.MQTT.TRANSMIT.QUEUE is to make sending messages directly to MQTT clients easier. Without altering the default transmission queue you must add a remote queue definition for every client that receives IBM WebSphere MQ messages; see "Sending a message to a client directly" on page 120.

4. Follow a procedure in "Authorizing MQTT clients to access WebSphere MQ objects" on page 122 to create one or more user IDs. The user IDs have the authority to publish, subscribe, and send publications to MQTT clients.
5. Install the telemetry (MQXR) service

```
type
installMQXRService_win.mqsc | runmqsc qMgr
```

6. Start the service

```
echo START SERVICE(SYSTEM.MQXR.SERVICE) | runmqsc qMgr
```

   The telemetry (MQXR) service is started automatically when the queue manager is started.

   It is started manually in this task, because the queue manager is already running.
7. Using IBM WebSphere MQ Explorer, configure telemetry channels to accept connections from MQTT clients.

   The telemetry channels must be configured such that their identities are one of the user IDs defined in step 4.

   See also DEFINE CHANNEL (MQTT).
8. Verify the configuration by running the sample client.

   For the sample client to work with your telemetry channel, the channel must authorize the client to publish, subscribe, and receive publications. The sample client connects to the telemetry channel on port 1883 by default. See also IBM WebSphere MQ Telemetry sample programs.

Creating SYSTEM.MQXR.SERVICE manually

Figure 21 on page 118 shows the **runmqsc** command to create the SYSTEM.MQXR.SERVICE manually on Windows.

```
DEF    SERVICE(SYSTEM.MQXR.SERVICE) +
    CONTROL(QMGR) +
    DESCR('Manages clients using MQXR protocols such as MQTT') +
    SERVTYPE(SERVER) +
    STARTCMD('+MQ_INSTALL_PATH+\mqxr\bin\runMQXRService.bat') +
    STARTARG('-m +QMNAME+  -d "+MQ_Q_MGR_DATA_PATH+\." -g "+MQ_DATA_PATH+\."') +
    STOPCMD('+MQ_INSTALL_PATH+\mqxr\bin\endMQXRService.bat') +
    STOPARG('-m +QMNAME+') +
    STDOUT('+MQ_Q_MGR_DATA_PATH+\mqxr.stdout') +
    STDERR('+MQ_Q_MGR_DATA_PATH+\mqxr.stderr')
```

Figure 21. installMQXRService_win.mqsc

Configure distributed queuing to send messages to MQTT clients

IBM WebSphere MQ applications can send MQTT v3 clients messages by publishing to subscription created by a client, or by sending a message directly. Whichever method is used, the message is placed on SYSTEM.MQTT.TRANSMIT.QUEUE, and sent to the client by the telemetry (MQXR) service. There are a number of ways to place a message on SYSTEM.MQTT.TRANSMIT.QUEUE.

Publishing a message in response to an MQTT client subscription

The telemetry (MQXR) service creates a subscription on behalf of the MQTT client. The client is the destination for any publications that match the subscription sent by the client. The telemetry services forwards matching publications back to the client.

An MQTT client is connected to WebSphere MQ as a queue manager, with its queue manager name set to its `ClientIdentifier`. The destination for publications to be sent to the client is a transmission queue, `SYSTEM.MQTT.TRANSMIT.QUEUE`. The telemetry service forwards messages on `SYSTEM.MQTT.TRANSMIT.QUEUE` to MQTT clients, using the target queue manager name as the key to a specific client.

The telemetry (MQXR) service opens the transmission queue using `ClientIdentifier` as the queue manager name. The telemetry (MQXR) service passes the object handle of the queue to the `MQSUB` call, to forward publications that match the client subscription. In the object name resolution, the `ClientIdentifier` is created as the remote queue manager name, and the transmission queue must resolve to `SYSTEM.MQTT.TRANSMIT.QUEUE`. Using standard WebSphere MQ object name resolution, *ClientIdentifier* is resolved as follows; see Table 6 on page 119.

1. *ClientIdentifier* matches nothing.

   ClientIdentifier is a remote queue manager name. It does not match the local queue manager name, a queue manager alias, or a transmission queue name.
   The queue name is not defined. Currently, the telemetry (MQXR) service sets `SYSTEM.MQTT.PUBLICATION.QUEUE` as the name of the queue. An MQTT v3 client does not support queues, so the resolved queue name is ignored by the client.
   The local queue manager property, `Default transmission queue`, name must be set to `SYSTEM.MQTT.TRANSMIT.QUEUE`, so that the publication is put on `SYSTEM.MQTT.TRANSMIT.QUEUE` to be sent to the client.
2. *ClientIdentifier* matches a queue manager alias named *ClientIdentifier*.

   ClientIdentifier is a remote queue manager name. It matches the name of a queue manager alias.
   The queue manager alias must be defined with *ClientIdentifier* as the remote queue manager name.
   By setting the transmission queue name in the queue manager alias definition it is not necessary for the default transmission to be set to `SYSTEM.MQTT.TRANSMIT.QUEUE`.

Table 6. Name resolution of an MQTT queue manager alias

	Input		Output		
ClientIdentifier	**Queue manager name**	**Queue name**	**Queue manager name**	**Queue name**	**Transmission queue**
Matches nothing	*ClientIdentifier*	*undefined*	*ClientIdentifier*	*undefined*	Default transmission queue. `SYSTEM.MQTT.TRANSMIT.QUEUE`
Matches a queue manager alias named *ClientIdentifier*	*ClientIdentifier*	*undefined*	*ClientIdentifier*	*undefined*	`SYSTEM.MQTT.TRANSMIT.QUEUE`

For further information about name resolution, see Name resolution.

Any WebSphere MQ program can publish to the same topic. The publication is sent to its subscribers, including MQTT v3 clients that have a subscription to the topic.

If an administrative topic is created in a cluster, with the attribute `CLUSTER(`*`clusterName`*`)`, any application in the cluster can publish to the client; for example:

```
echo DEFINE TOPIC('MQTTExamples') TOPICSTR('MQTT Examples') CLUSTER(MQTT) REPLACE | runmqsc qMgr
```

Figure 22. Defining a cluster topic on Windows

Note: Do not give `SYSTEM.MQTT.TRANSMIT.QUEUE` a cluster attribute.

MQTT client subscribers and publishers can connect to different queue managers. The subscribers and publishers can be part of the same cluster, or connected by a publish/subscribe hierarchy. The publication is delivered from the publisher to the subscriber using WebSphere MQ.

Sending a message to a client directly

An alternative to a client creating a subscription and receiving a publication that matches the subscription topic, send a message to an MQTT v3 client directly. MQTT V3 client applications cannot send messages directly, but other application, such as WebSphere MQ applications, can.

The WebSphere MQ application must know the `ClientIdentifier` of the MQTT v3 client. As MQTT v3 clients do not have queues, the target queue name is passed to the MQTT v3 application client `messageArrived` method as a topic name. For example, in an MQI program, create an object descriptor with the client as the `ObjectQmgrName`:

```
MQOD.ObjectQmgrName = ClientIdentifier;
MQOD.ObjectName = name;
```

Figure 23. MQI Object descriptor to send a message to an MQTT v3 client destination

If the application is written using JMS, create a point-to-point destination; for example:

```
javax.jms.Destination jmsDestination =
                      (javax.jms.Destination)jmsFactory.createQueue
                      ("queue://ClientIdentifier/name");
```

Figure 24. JMS destination to send a message to an MQTT v3 client

To send an unsolicited message to an MQTT client use a remote queue definition. The remote queue manager name must resolvedto the `ClientIdentifier` of the client. The transmission queue must resolve to `SYSTEM.MQTT.TRANSMIT.QUEUE`; see Table 7 on page 120. The remote queue name can be anything. The client receives it as a topic string.

Table 7. Name resolution of an MQTT client remote queue definition

Input		Output		
Queue name	**Queue manager name**	**Queue name**	**Queue manager name**	**Transmission queue**
Name of remote queue definition	Blank or local queue manager name	Remote queue name used as a topic string	*`ClientIdentifier`*	`SYSTEM.MQTT.TRANSMIT.QUEUE`

If the client is connected, the message is sent directly to the MQTT client, which calls the `messageArrived` method; see `messageArrived` method.

If the client has disconnected with a persistent session, the message is stored in `SYSTEM.MQTT.TRANSMIT.QUEUE`; see MQTT stateless and stateful sessions . It is forwarded to the client when the client reconnects to the session again.

If you send a non-persistent message it is sent to the client with "at most once" quality of service, `QoS=0`. If you send a persistent message directly to a client, by default, it is sent with "exactly once" quality of service, `QoS=2`. As the client might not have a persistence mechanism, the client can lower the quality of service it accepts for messages sent directly. To lower the quality of service for messages sent directly to a client, make a subscription to the topic `DEFAULT.QoS`. Specify the maximum quality of service the client can support.

MQTT client identification, authorization, and authentication

The telemetry (MQXR) service publishes, or subscribes to, WebSphere MQ topics on behalf of MQTT clients, using MQTT channels. The WebSphere MQ administrator configures the MQTT channel identity that is used for WebSphere MQ authorization. The administrator can define a common identity for the channel, or use the `Username` or `ClientIdentifier` of a client connected to the channel.

The telemetry (MQXR) service can authenticate the client using the `Username` supplied by the client, or by using a client certificate. The `Username` is authenticated using a password provided by the client.

To summarize: Client identification is the selection of the client identity. Depending on the context, the client is identified by the `ClientIdentifier`, `Username`, a common client identity created by the administrator, or a client certificate. The client identifier used for authenticity checking does not have to be the same identifier that is used for authorization.

MQTT client programs set the `Username` and `Password` that are sent to the server using an MQTT channel. They can also set the SSL properties that are required to encrypt and authenticate the connection. The administrator decides whether to authenticate the MQTT channel, and how to authenticate the channel.

To authorize an MQTT client to access WebSphere MQ objects, authorize the `ClientIdentifier`, or `Username` of the client, or authorize a common client identity. To permit a client to connect to WebSphere MQ, authenticate the `Username`, or use a client certificate. Configure JAAS to authenticate the `Username`, and configure SSL to authenticate a client certificate.

If you set a `Password` at the client, either encrypt the connection using VPN, or configure the MQTT channel to use SSL, to keep the password private.

It is difficult to manage client certificates. For this reason, if the risks associated with password authentication are acceptable, password authentication is often used to authenticate clients.

If there is a secure way to manage and store the client certificate it is possible to rely on certificate authentication. However, it is rarely the case that certificates can be managed securely in the types of environments that telemetry is used in. Instead, the authentication of devices using client certificates is complemented by authenticating client passwords at the server. Because of the additional complexity, the use of client certificates is restricted to highly sensitive applications. The use of two forms of authentication is called two-factor authentication. You must know one of the factors, such as a password, and have the other, such as a certificate.

In a highly sensitive application, such as a chip-and-pin device, the device is locked down during manufacture to prevent tampering with the internal hardware and software. A trusted, time-limited, client certificate is copied to the device. The device is deployed to the location where it is to be used. Further authentication is performed each time the device is used, either using a password, or another certificate from a smart card.

MQTT client identity and authorization

Use the `ClientIdentifier`, `Username`, or a common client identity for authorization to access WebSphere MQ objects.

The WebSphere MQ administrator has three choices for selecting the identity of the MQTT channel. The administrator makes the choice when defining or modifying the MQTT channel used by the client. The identity is used to authorize access to WebSphere MQ topics. The choices are:

1. The client identifier.
2. An identity the administrator provides for the channel.
3. The `Username` passed from the MQTT client.

`Username` is an attribute of the MqttConnectOptions class. It must be set before the client connects to the service. Its default value is null.

Use the WebSphere MQ **`setmqaut`** command to select which objects, and which actions, are authorized to be used by the identity associated with the MQTT channel. For example, to authorize a channel identity, `MQTTClient`, provided by the administrator of queue manager, `QM1`:

```
setmqaut -m QM1 -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p MQTTClient -all +put
setmqaut -m QM1 -t topic -n SYSTEM.BASE.TOPIC -p MQTTClient -all +pub +sub
```

Authorizing MQTT clients to access WebSphere MQ objects

Follow these steps to authorize MQTT clients to publish and subscribe to WebSphere MQ Objects. The steps follow four alternative access control patterns.

Before you begin

MQTT clients are authorized to access objects in WebSphere MQ by being assigned an identity when they connect to a telemetry channel. The WebSphere MQ Administrator configures the telemetry channel using WebSphere MQ Explorer to give a client one of three types of identity:

1. `ClientIdentifier`
2. `Username`
3. A name the administrator assigns to the channel.

Whichever type is used, the identity must be defined to WebSphere MQ as a principal by the installed authorization service. The default authorization service on Windows or Linux is called the Object Authority Manager (OAM). If you are using the OAM, the identity must be defined as a user ID.

Use the identity to give a client, or collection of clients, permission to publish or subscribe to topics defined in WebSphere MQ. If an MQTT client has subscribed to a topic, use the identity to give it permission to receive the resulting publications.

It is hard to manage a system with tens of thousands of MQTT clients, each requiring individual access permissions. One solution is to define common identities, and associate individual MQTT clients with one of the common identities. Define as many common identities as you require to define different combinations of permissions. Another solution is to write your own authorization service that can deal more easily with thousands of users than the operating system.

You can combine MQTT clients into common identities in two ways, using the OAM:

1. Define multiple telemetry channels, each with a different user ID that the administrator allocates using WebSphere MQ Explorer. Clients connecting using different TCP/IP port numbers are associated with different telemetry channels, and are assigned different identities.
2. Define a single telemetry channel, but have each client select a `Username` from a small set of user IDs. The administrator configures the telemetry channel to select the client `Username` as its identity.

In this task, the identity of the telemetry channel is called *`mqttUser`*, regardless of how it is set. If collections of clients use different identities, use multiple *`mqttUsers`*, one for each collection of clients. As the task uses the OAM, each *`mqttUser`* must be a user ID.

About this task

In this task, you have a choice of four access control patterns that you can tailor to specific requirements. The patterns differ in their granularity of access control.

- "No access control" on page 123
- "Coarse-grained access control" on page 123
- "Medium-grained access control" on page 123
- "Fine-grained access control" on page 124

The result of the models is to assign *mqttUsers* sets of permissions to publish and subscribe to WebSphere MQ, and receive publications from WebSphere MQ.

No access control
MQTT clients are given WebSphere MQ administrative authority, and can perform any action on any object.

Procedure

1. Create a user ID *mqttUser* to act as the identity of all MQTT clients.
2. Add *mqttUser* to the mqm group; see Adding a user to a group on Windows, or Adding a user to a group on Linux

Coarse-grained access control
MQTT clients have authority to publish and subscribe, and to send messages to MQTT clients. They do not have authority to perform other actions, or to access other objects.

Procedure

1. Create a user ID *mqttUser* to act as the identity of all MQTT clients.
2. Authorize *mqttUser* to publish and subscribe to all topics and to send publications to MQTT clients.

```
setmqaut -m qMgr -t topic -n SYSTEM.BASE.TOPIC -p mqttUser -all +pub +sub
setmqaut -m qMgr -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p mqttUser -all +put
```

Medium-grained access control
MQTT clients are divided into different groups to publish and subscribe to different sets of topics, and to send messages to MQTT clients.

Procedure

1. Create multiple user IDs, *mqttUsers*, and multiple administrative topics in the publish/subscribe topic tree.
2. Authorize different *mqttUsers* to different topics.

```
setmqaut -m qMgr -t topic -n topic1 -p mqttUserA -all +pub +sub
setmqaut -m qMgr -t topic -n topic2 -p mqttUserB -all +pub +sub
```

3. Create a group *mqtt*, and add all *mqttUsers* to the group.
4. Authorize *mqtt* to send topics to MQTT clients.

```
setmqaut -m qMgr -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p mqtt -all +put
```

Fine-grained access control
MQTT clients are incorporated into an existing system of access control, that authorizes groups to perform actions on objects.

About this task

A user ID is assigned to one or more operating system groups depending on the authorizations it requires. If WebSphere MQ applications are publishing and subscribing to the same topic space as MQTT clients, use this model. The groups are referred to as `Publish`*X*, `Subscribe`*Y*, and `mqtt`

`Publish`*X*
Members of `Publish`*X* groups can publish to *`topicX`*.

`Subscribe`*Y*
Members of `Subscribe`*Y* groups can subscribe to *`topicY`*.

`mqtt`
Members of the *`mqtt`* group can send publications to MQTT clients.

Procedure

1. Create multiple groups, `Publish`*X* and `Subscribe`*Y* that are allocated to multiple administrative topics in the publish/subscribe topic tree.
2. Create a group `mqtt`.
3. Create multiple user IDs, *`mqttUsers`*, and add the users to any of the groups, depending on what they are authorized to do.
4. Authorize different `Publish`*X* and `Subscribe`*X* groups to different topics, and authorize the *`mqtt`* group to send messages to MQTT clients.

```
setmqaut -m qMgr -t topic -n topic1 -p PublishX -all +pub
setmqaut -m qMgr -t topic -n topic1 -p SubscribeX -all +pub +sub
setmqaut -m qMgr -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p mqtt -all +put
```

MQTT client authentication using a password

Authenticate the `Username` using the client password. You can authenticate the client using a different identity to the identity used to authorize the client to publish and subscribe to topics.

The telemetry (MQXR) service uses JAAS to authenticate the client `Username`. JAAS uses the `Password` supplied by the MQTT client.

The WebSphere MQ administrator decides whether to authenticate the `Username`, or not to authenticate at all, by configuring the MQTT channel a client connects to. Clients can be assigned to different channels, and each channel can be configured to authenticate its clients in different ways. Using JAAS, you can configure which methods must authenticate the client, and which can optionally authenticate the client.

The choice of identity for authentication does not affect the choice of identity for authorization. You might want to set up a common identity for authorization for administrative convenience, but authenticate each user to use that identity. The following procedure outlines the steps to authenticate individual users to use a common identity:

1. The WebSphere MQ administrator sets the MQTT channel identity to any name, such as `MQTTClientUser`, using WebSphere MQ Explorer.
2. The WebSphere MQ administrator authorizes `MQTTClient` to publish and subscribe to any topic:

```
setmqaut -m QM1 -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p MQTTClient -all +put
setmqaut -m QM1 -t topic -n SYSTEM.BASE.TOPIC -p MQTTClient -all +pub +sub
```

3. The MQTT client application developer creates an `MqttConnectOptions` object and sets `Username` and `Password` before connecting to the server.
4. The security developer creates a JAAS `LoginModule` to authenticate the `Username` with the `Password` and includes it in the JAAS configuration file.

5. The WebSphere MQ administrator configures the MQTT channel to authenticate the `UserName` of the client using JAAS.

MQTT client authentication using SSL

Connections between the MQTT client and the queue manager are always initiated by the MQTT client. The MQTT client is always the SSL client. Client authentication of the server and server authentication of the MQTT client are both optional.

By providing the client with a private signed digital certificate, you can authenticate the MQTT client to IBM WebSphere MQ. The IBM WebSphere MQ Administrator can force MQTT clients to authenticate themselves to the queue manager using SSL. You can only request client authentication as part of mutual authentication.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

Client authentication using SSL relies upon the client having a secret. The secret is the private key of the client in the case of a self-signed certificate, or a key provided by a certificate authority. The key is used to sign the digital certificate of the client. Anyone in possession of the corresponding public key can verify the digital certificate. Certificates can be trusted, or if they are chained, traced back through a certificate chain to a trusted root certificate. Client verification sends all the certificates in the certificate chain provided by the client to the server. The server checks the certificate chain until it finds a certificate it trusts. The trusted certificate is either the public certificate generated from a self-signed certificate, or a root certificate typically issued by a certificate authority. As a final, optional, step the trusted certificate can be compared with a "live" certificate revocation list.

The trusted certificate might be issued by a certificate authority and already included in the JRE certificate store. It might be a self-signed certificate, or any certificate that has been added to the telemetry channel keystore as a trusted certificate.

Note: The telemetry channel has a combined keystore/truststore that holds both the private keys to one or more telemetry channels, and any public certificates needed to authenticate clients. Because an SSL channel must have a keystore, and it is the same file as the channel truststore, the JRE certificate store is never referenced. The implication is that if authentication of a client requires a CA root certificate, you must place the root certificate in the keystore for the channel, even if the CA root certificate is already in the JRE certificate store. The JRE certificate store is never referenced.

Think about the threats that client authentication is intended to counter, and the roles the client and server play in countering the threats. Authenticating the client certificate alone is insufficient to prevent unauthorized access to a system. If someone else has got hold of the client device, the client device is not necessarily acting with the authority of the certificate holder. Never rely on a single defense against unwanted attacks. At least use a two-factor authentication approach and supplement possession of a certificate with knowledge of private information. For example, use JAAS, and authenticate the client using a password issued by the server.

The primary threat to the client certificate is that it gets into the wrong hands. The certificate is held in a password protected keystore at the client. How does it get placed in the keystore? How does the MQTT client get the password to the keystore? How secure is the password protection? Telemetry devices are often easy to remove, and then can be hacked in private. Must the device hardware be tamper-proof? Distributing and protecting client-side certificates is recognized to be hard; it is called the key-management problem.

A secondary threat is that the device is misused to access servers in unintended ways. For example, if the MQTT application is tampered with, it might be possible to use a weakness in the server configuration using the authenticated client identity.

To authenticate an MQTT client using SSL, configure the telemetry channel, and the client.

•

•

Telemetry channel configuration for MQTT client authentication using SSL

The IBM WebSphere MQ administrator configures telemetry channels at the server. Each channel is configured to accept a TCP/IP connection on a different port number. SSL channels are configured with passphrase protected access to key files. If an SSL channel is defined with no passphrase or key file, the channel does not accept SSL connections.

Set the property, `com.ibm.mq.MQTT.ClientAuth` of an SSL telemetry channel to `REQUIRED` to force all clients connecting on that channel to provide proof that they have verified digital certificates. The client certificates are authenticated using certificates from certificate authorities, leading to a trusted root certificate. If the client certificate is self-signed, or is signed by a certificate that is from a certificate authority, the publicly signed certificates of the client, or certificate authority, must be stored securely at the server.

Place the publicly signed client certificate or the certificate from the certificate authority in the telemetry channel keystore. At the server, publicly signed certificates are stored in the same key file as privately signed certificates, rather than in a separate truststore.

The server verifies the signature of any client certificates it is sent using all the public certificates and cipher suites it has. The server verifies the key chain. The queue manager can be configured to test the certificate against the certificate revocation list. The queue manager revocation namelist property is `SSLCRLNL`.

If any of the certificates a client sends is verified by a certificate in the server keystore, then the client is authenticated.

The WebSphere MQ administrator can configure the same telemetry channel to use JAAS to check the `UserName` or `ClientIdentifier` of the client with the client `Password`.

You can use the same keystore for multiple telemetry channels.

Verification of at least one digital certificate in the password protected client keystore on the device authenticates the client to the server. The digital certificate is only used for authentication by WebSphere MQ. It is not used to verify the TCP/IP address of the client, or set the identity of the client for authorization or accounting. The identity of the client adopted by the server is either the `Username` or `ClientIdentifier` of the client, or an identity created by the WebSphere MQ administrator.

You can also use SSL cipher suites for client authentication. Here is an alphabetic list of the SSL cipher suites that are currently supported:

- `SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DH_anon_EXPORT_WITH_RC4_40_MD5`
- `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`
- `SSL_DH_anon_WITH_AES_128_CBC_SHA`
- `SSL_DH_anon_WITH_DES_CBC_SHA`
- `SSL_DH_anon_WITH_RC4_128_MD5`
- `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
- `SSL_DHE_DSS_WITH_AES_128_CBC_SHA`
- `SSL_DHE_DSS_WITH_DES_CBC_SHA`
- `SSL_DHE_DSS_WITH_RC4_128_SHA`
- `SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA`
- `SSL_DHE_RSA_WITH_AES_128_CBC_SHA`
- `SSL_DHE_RSA_WITH_DES_CBC_SHA`
- `SSL_KRB5_EXPORT_WITH_DES_CBC_40_MD5`

- SSL_KRB5_EXPORT_WITH_DES_CBC_40_SHA
- SSL_KRB5_EXPORT_WITH_RC4_40_MD5
- SSL_KRB5_EXPORT_WITH_RC4_40_SHA
- SSL_KRB5_WITH_3DES_EDE_CBC_MD5
- SSL_KRB5_WITH_3DES_EDE_CBC_SHA
- SSL_KRB5_WITH_DES_CBC_MD5
- SSL_KRB5_WITH_DES_CBC_SHA
- SSL_KRB5_WITH_RC4_128_MD5
- SSL_KRB5_WITH_RC4_128_SHA
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
- V7.5.0.2 SSL_RSA_FIPS_WITH_AES_128_CBC_SHA256
- V7.5.0.2 SSL_RSA_FIPS_WITH_AES_256_CBC_SHA256
- SSL_RSA_FIPS_WITH_DES_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_AES_128_CBC_SHA
- V7.5.0.2 SSL_RSA_WITH_AES_128_CBC_SHA256
- V7.5.0.2 SSL_RSA_WITH_AES_256_CBC_SHA256
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_NULL_SHA
- V7.5.0.2 SSL_RSA_WITH_NULL_SHA256
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA

V7.5.0.2 If you plan to use SHA-2 cipher suites, see System requirements for using SHA-2 cipher suites with MQTT channels.

Related concepts

"Telemetry channel configuration for channel authentication using SSL" on page 128
The IBM WebSphere MQ administrator configures telemetry channels at the server. Each channel is configured to accept a TCP/IP connection on a different port number. SSL channels are configured with passphrase protected access to key files. If an SSL channel is defined with no passphrase or key file, the channel does not accept SSL connections.

CipherSpecs and CipherSuites

Related reference

DEFINE CHANNEL (MQTT)
ALTER CHANNEL (MQTT)

Telemetry channel authentication using SSL

Connections between the MQTT client and the queue manager are always initiated by the MQTT client. The MQTT client is always the SSL client. Client authentication of the server and server authentication of the MQTT client are both optional.

The client always attempts to authenticate the server, unless the client is configured to use a CipherSpec that supports anonymous connection. If the authentication fails, then the connection is not established.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

Server authentication using SSL authenticates the server to which you are about to send confidential information to. The client performs the checks matching the certificates sent from the server, against certificates placed in its truststore, or in its JRE `cacerts` store.

The JRE certificate store is a JKS file, `cacerts`. It is located in `JRE InstallPath\lib\security\`. It is installed with the default password `changeit`. You can either store certificates you trust in the JRE certificate store, or in the client truststore. You cannot use both stores. Use the client truststore if you want to keep the public certificates the client trusts separate from certificates other Java applications use. Use the JRE certificate store if you want to use a common certificate store for all Java applications running on the client. If you decide to use the JRE certificate store review the certificates it contains, to make sure you trust them.

You can modify the JSSE configuration by supplying a different trust provider. You can customize a trust provider to perform different checks on a certificate. In some OGSi environments that have used the MQTT client, the environment provides a different trust provider.

To authenticate the telemetry channel using SSL, configure the server, and the client.

•

•

Telemetry channel configuration for channel authentication using SSL

The IBM WebSphere MQ administrator configures telemetry channels at the server. Each channel is configured to accept a TCP/IP connection on a different port number. SSL channels are configured with passphrase protected access to key files. If an SSL channel is defined with no passphrase or key file, the channel does not accept SSL connections.

Store the digital certificate of the server, signed with its private key, in the keystore that the telemetry channel is going to use at the server. Store any certificates in its key chain in the keystore, if you want to transmit the key chain to the client. Configure the telemetry channel using WebSphere MQ explorer to use SSL. Provide it with the path to the keystore, and the passphrase to access the keystore. If you do not set the TCP/IP port number of the channel, the SSL telemetry channel port number defaults to 8883.

You can also use SSL cipher suites for channel authentication. Here is an alphabetic list of the SSL cipher suites that are currently supported:

- `SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DH_anon_EXPORT_WITH_RC4_40_MD5`
- `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`
- `SSL_DH_anon_WITH_AES_128_CBC_SHA`
- `SSL_DH_anon_WITH_DES_CBC_SHA`
- `SSL_DH_anon_WITH_RC4_128_MD5`
- `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
- `SSL_DHE_DSS_WITH_AES_128_CBC_SHA`
- `SSL_DHE_DSS_WITH_DES_CBC_SHA`
- `SSL_DHE_DSS_WITH_RC4_128_SHA`
- `SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA`

- `SSL_DHE_RSA_WITH_AES_128_CBC_SHA`
- `SSL_DHE_RSA_WITH_DES_CBC_SHA`
- `SSL_KRB5_EXPORT_WITH_DES_CBC_40_MD5`
- `SSL_KRB5_EXPORT_WITH_DES_CBC_40_SHA`
- `SSL_KRB5_EXPORT_WITH_RC4_40_MD5`
- `SSL_KRB5_EXPORT_WITH_RC4_40_SHA`
- `SSL_KRB5_WITH_3DES_EDE_CBC_MD5`
- `SSL_KRB5_WITH_3DES_EDE_CBC_SHA`
- `SSL_KRB5_WITH_DES_CBC_MD5`
- `SSL_KRB5_WITH_DES_CBC_SHA`
- `SSL_KRB5_WITH_RC4_128_MD5`
- `SSL_KRB5_WITH_RC4_128_SHA`
- `SSL_RSA_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_RSA_EXPORT_WITH_RC4_40_MD5`
- `SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA`
- V 7.5.0.2 `SSL_RSA_FIPS_WITH_AES_128_CBC_SHA256`
- V 7.5.0.2 `SSL_RSA_FIPS_WITH_AES_256_CBC_SHA256`
- `SSL_RSA_FIPS_WITH_DES_CBC_SHA`
- `SSL_RSA_WITH_3DES_EDE_CBC_SHA`
- `SSL_RSA_WITH_AES_128_CBC_SHA`
- V 7.5.0.2 `SSL_RSA_WITH_AES_128_CBC_SHA256`
- V 7.5.0.2 `SSL_RSA_WITH_AES_256_CBC_SHA256`
- `SSL_RSA_WITH_DES_CBC_SHA`
- `SSL_RSA_WITH_NULL_MD5`
- `SSL_RSA_WITH_NULL_SHA`
- V 7.5.0.2 `SSL_RSA_WITH_NULL_SHA256`
- `SSL_RSA_WITH_RC4_128_MD5`
- `SSL_RSA_WITH_RC4_128_SHA`

V 7.5.0.2 If you plan to use SHA-2 cipher suites, see System requirements for using SHA-2 cipher suites with MQTT channels.

Related concepts

"Telemetry channel configuration for MQTT client authentication using SSL" on page 126
The IBM WebSphere MQ administrator configures telemetry channels at the server. Each channel is configured to accept a TCP/IP connection on a different port number. SSL channels are configured with passphrase protected access to key files. If an SSL channel is defined with no passphrase or key file, the channel does not accept SSL connections.

CipherSpecs and CipherSuites

Related reference

DEFINE CHANNEL (MQTT)

ALTER CHANNEL (MQTT)

Publication privacy on telemetry channels

The privacy of MQTT publications sent in either direction across telemetry channels is secured by using SSL to encrypt transmissions over the connection.

MQTT clients that connect to telemetry channels use SSL to secure the privacy of publications transmitted on the channel using symmetric key cryptography. Because the endpoints are not authenticated, you cannot trust channel encryption alone. Combine securing privacy with server or mutual authentication.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

For a typical configuration, which encrypts the channel and authenticates the server, consult "Telemetry channel authentication using SSL" on page 127.

Encrypting SSL connections without authenticating the server exposes the connection to man-in-the-middle attacks. Although the information you exchange is protected against eavesdropping, you do not know who you are exchanging it with. Unless you control the network, you are exposed to someone intercepting your IP transmissions, and masquerading as the endpoint.

You can create an encrypted SSL connection, without authenticating the server, by using a Diffie-Hellman key exchange CipherSpec that supports anonymous SSL. The master secret, shared between the client and server, and used to encrypt SSL transmissions, is established without exchanging a privately signed server certificate.

Because anonymous connections are insecure, most SSL implementations do not default to using anonymous CipherSpecs. If a client request for SSL connection is accepted by a telemetry channel, the channel must have a keystore protected by a passphrase. By default, since SSL implementations do not use anonymous CipherSpecs, the keystore must contain a privately signed certificate that the client can authenticate.

If you use anonymous CipherSpecs, the server keystore must exist, but it need not contain any privately signed certificates.

Another way to establish an encrypted connection is to replace the trust provider at the client with your own implementation. Your trust provider would not authenticate the server certificate, but the connection would be encrypted.

SSL configuration of MQTT clients and telemetry channels

MQTT clients and the WebSphere MQ Telemetry (MQXR) service use Java Secure Socket Extension (JSSE) to connect telemetry channels using SSL. The IBM WebSphere MQ Telemetry daemon for devices does not support SSL.

Configure SSL to authenticate the telemetry channel, the MQTT client, and encrypt the transfer of messages between clients and the telemetry channel.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

You can configure the connection between a Java MQTT client and a telemetry channel to use the SSL protocol over TCP/IP. What is secured depends on how you configure SSL to use JSSE. Starting with the most secured configuration, you can configure three different levels of security:

1. Permit only trusted MQTT clients to connect. Connect an MQTT client only to a trusted telemetry channel. Encrypt messages between the client and the queue manager; see "MQTT client authentication using SSL" on page 125
2. Connect an MQTT client only to a trusted telemetry channel. Encrypt messages between the client and the queue manager; see "Telemetry channel authentication using SSL" on page 127.

3. Encrypt messages between the client and the queue manager; see "Publication privacy on telemetry channels " on page 130.

JSSE configuration parameters

Modify JSSE parameters to alter the way an SSL connection is configured. The JSSE configuration parameters are arranged into three sets:

1. IBM WebSphere MQ Telemetry channel
2. MQTT Java client
3. JRE

Configure the telemetry channel parameters using IBM WebSphere MQ Explorer. Set the MQTT Java Client parameters in the `MqttConnectionOptions.SSLProperties` attribute. Modify JRE security parameters by editing files in the JRE security directory on both the client and server.

IBM WebSphere MQ Telemetry channel

Set all the telemetry channel SSL parameters using WebSphere MQ Explorer.

`ChannelName`

`ChannelName` is a required parameter on all channels.

The channel name identifies the channel associated with a particular port number. Name channels to help you administer sets of MQTT clients.

`PortNumber`

`PortNumber` is an optional parameter on all channels. It defaults to 1883 for TCP channels, and 8883 for SSL channels.

The TCP/IP port number associated with this channel. MQTT clients are connected to a channel by specifying the port defined for the channel. If the channel has SSL properties, the client must connect using the SSL protocol; for example:

```
MQTTClient mqttClient = new MqttClient( "ssl://www.example.org:8884", "clientId1");
mqttClient.connect();
```

`KeyFileName`

`KeyFileName` is a required parameter for SSL channels. It must be omitted for TCP channels.

`KeyFileName` is the path to the Java keystore containing digital certificates that you provide. Use JKS, JCEKS or PKCS12 as the type of keystore on the server.

Identify the keystore type by using one of the following file extensions:

`.jks`
`.jceks`
`.p12`
`.pkcs12`

A keystore with any other file extension is assumed to be a JKS keystore.

You can combine one type of keystore at the server with other types of keystore at the client.

Place the private certificate of the server in the keystore. The certificate is known as the server certificate. The certificate can be self-signed, or part of a certificate chain that is signed by a signing authority.

If you are using a certificate chain, place the associated certificates in the server keystore.

The server certificate, and any certificates in its certificate chain, are sent to clients to authenticate the identity of the server.

If you have set `ClientAuth` to `Required`, the keystore must contain any certificates necessary to authenticate the client. The client sends a self-signed certificate, or a certificate chain, and the client is authenticated by the first verification of this material against a certificate in the keystore. Using a certificate chain, one certificate can verify many clients, even if they are issued with different client certificates.

`PassPhrase`

`PassPhrase` is a required parameter for SSL channels. It must be omitted for TCP channels.

The passphrase is used to protect the keystore.

`ClientAuth`

`ClientAuth` is an optional SSL parameter. It defaults to no client authentication. It must be omitted for TCP channels.

Set `ClientAuth` if you want the telemetry (MQXR) service to authenticate the client, before permitting the client to connect to the telemetry channel.

If you set `ClientAuth`, the client must connect to the server using SSL, and authenticate the server. In response to setting `ClientAuth`, the client sends its digital certificate to the server, and any other certificates in its keystore. Its digital certificate is known as the client certificate. These certificates are authenticated against certificates held in the channel keystore, and in the JRE `cacerts` store.

`CipherSuite`

`CipherSuite` is an optional SSL parameter. It defaults to try all the enabled CipherSpecs. It must be omitted for TCP channels.

If you want to use a particular CipherSpec, set `CipherSuite` to the name of the CipherSpec that must be used to establish the SSL connection.

The telemetry service and MQTT client negotiate a common CipherSpec from all the CipherSpecs that are enabled at each end. If a specific CipherSpec is specified at either or both ends of the connection, it must match the CipherSpec at the other end.

Install additional ciphers by adding additional providers to JSSE.

`Federal Information Processing Standards (FIPS)`

FIPS is an optional setting. By default it is not set.

Either in the properties panel of the queue manager, or using **`runmqsc`**, set `SSLFIPS`. `SSLFIPS` specifies whether only FIPS-certified algorithms are to be used.

`Revocation namelist`

Revocation namelist is an optional setting. By default it is not set.

Either in the properties panel of the queue manager, or using **`runmqsc`**, set `SSLCRLNL`. `SSLCRLNL` specifies a namelist of authentication information objects which are used to provide certificate revocation locations.

No other queue manager parameters that set SSL properties are used.

MQTT Java client

Set SSL properties for the Java client in `MqttConnectionOptions.SSLProperties`; for example:

```
java.util.Properties sslClientProperties = new Properties();
sslClientProperties.setProperty("com.ibm.ssl.keyStoreType", "JKS");
com.ibm.micro.client.mqttv3.MqttConnectOptions conOptions = new MqttConnectOptions();
conOptions.setSSLProperties(sslClientProperties);
```

The names and values of specific properties are described in the `MqttConnectOptions` class. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

`Protocol`

`Protocol` is optional.

The protocol is selected in negotiation with the telemetry server. If you require a specific protocol you can select one. If the telemetry server does not support the protocol the connection fails.

`ContextProvider`

`ContextProvider` is optional.

`KeyStore`

`KeyStore` is optional. Configure it if `ClientAuth` is set at the server to force authentication of the client.

Place the digital certificate of the client, signed using its private key, into the keystore. Specify the keystore path and password. The type and provider are optional. JKS is the default type, and IBMJCE is the default provider.

Specify a different keystore provider to reference a class that adds a new keystore provider. Pass the name of the algorithm used by the keystore provider to instantiate the `KeyManagerFactory` by setting the key manager name.

`TrustStore`

`TrustStore` is optional. You can place all the certificates you trust in the JRE `cacerts` store.

Configure the truststore if you want to have a different truststore for the client. You might not configure the truststore if the server is using a certificate issued by a well known CA that already has its root certificate stored in `cacerts`.

Add the publicly signed certificate of the server or the root certificate to the truststore, and specify the truststore path and password. JKS is the default type, and IBMJCE is the default provider.

Specify a different truststore provider to reference a class that adds a new truststore provider. Pass the name of the algorithm used by the truststore provider to instantiate the `TrustManagerFactory` by setting the trust manager name.

JRE

Other aspects of Java security that affect the behavior of SSL on both the client and server are configured in the JRE. The configuration files on Windows are in *`Java Installation Directory`*`\jre\lib\security`. If you are using the JRE shipped with IBM WebSphere MQ the path is as shown in the following table:

Table 8. Filepaths by platform for JRE SSL configuration files

Platform	**Filepath**
Windows	*`WMQ Installation Directory`*`\java\jre\lib\security`
Other UNIX and Linux platforms	*`WMQ Installation Directory`*`/java/jre64/jre/lib/security`

Well-known certificate authorities

The `cacerts` file contains the root certificates of well-known certificate authorities. The `cacerts` is used by default, unless you specify a truststore. If you use the `cacerts` store, or do not provide a truststore, you must review and edit the list of signers in `cacerts` to meet your security requirements.

You can open `cacerts` using the WebSphere MQ command `strmqikm`.which runs the IBM Key Management utility. Open `cacerts` as a JKS file, using the password `changeit`. Modify the password to secure the file.

Configuring security classes

Use the `java.security` file to register additional security providers and other default security properties.

Permissions

Use the `java.policy` file to modify the permissions granted to resources. `javaws.policy` grants permissions to `javaws.jar`

Encryption strength

Some JREs ship with reduced strength encryption. If you cannot import keys into keystores, reduced strength encryption might be the cause. Either, try starting **`ikeyman`** using the **`strmqikm`** command, or download strong, but limited jurisdiction files from IBM developer kits, Security information.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country. Check its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Modify the trust provider to permit the client to connect to any server

The example illustrates how to add a trust provider and reference it from the MQTT client code. The example performs no authentication of the client or server. The resulting SSL connection is encrypted without being authenticated.

The code snippet in Figure 25 on page 134 sets the `AcceptAllProviders` trust provider and trust manager for the MQTT client.

```
java.security.Security.addProvider(new AcceptAllProvider());
java.util.Properties sslClientProperties = new Properties();
sslClientProperties.setProperty("com.ibm.ssl.trustManager","TrustAllCertificates");
sslClientProperties.setProperty("com.ibm.ssl.trustStoreProvider","AcceptAllProvider");
conOptions.setSSLProperties(sslClientProperties);
```

Figure 25. MQTT Client code snippet

```
package com.ibm.mq.id;
public class AcceptAllProvider extends java.security.Provider {
    private static final long serialVersionUID = 1L;
    public AcceptAllProvider() {
        super("AcceptAllProvider", 1.0, "Trust all X509 certificates");
        put("TrustManagerFactory.TrustAllCertificates",
                AcceptAllTrustManagerFactory.class.getName());
    }
```

Figure 26. AcceptAllProvider.java

```
    protected static class AcceptAllTrustManagerFactory extends
            javax.net.ssl.TrustManagerFactorySpi {
        public AcceptAllTrustManagerFactory() {}
        protected void engineInit(java.security.KeyStore keystore) {}
        protected void engineInit(
                javax.net.ssl.ManagerFactoryParameters parameters) {}
        protected javax.net.ssl.TrustManager[] engineGetTrustManagers() {
            return new javax.net.ssl.TrustManager[] { new AcceptAllX509TrustManager() };
        }
```

Figure 27. AcceptAllTrustManagerFactory.java

```
    protected static class AcceptAllX509TrustManager implements
            javax.net.ssl.X509TrustManager {
        public void checkClientTrusted(
                java.security.cert.X509Certificate[] certificateChain,
                String authType) throws java.security.cert.CertificateException {
            report("Client authtype=" + authType);
            for (java.security.cert.X509Certificate certificate : certificateChain) {
                report("Accepting:" + certificate);
            }
        }
        public void checkServerTrusted(
                java.security.cert.X509Certificate[] certificateChain,
                String authType) throws java.security.cert.CertificateException {
            report("Server authtype=" + authType);
            for (java.security.cert.X509Certificate certificate : certificateChain) {
                report("Accepting:" + certificate);
            }
        }
        public java.security.cert.X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[0];
        }
    private static void report(String string) {
           System.out.println(string);
      }
    }
```

Figure 28. AcceptAllX509TrustManager.java

Telemetry channel JAAS configuration

Configure JAAS to authenticate the `Username` sent by the client.

The WebSphere MQ administrator configures which MQTT channels require client authentication using JAAS. Specify the name of a JAAS configuration for each channel that is to perform JAAS authentication. Channels can all use the same JAAS configuration, or they can use different JAAS configurations. The configurations are defined in *`WMQData directory`*`\qmgrs\`*`qMgrName`*`\mqxr\jaas.config`.

The `jaas.config` file is organized by JAAS configuration name. Under each configuration name is a list of Login configurations; see Figure 29 on page 136.

JAAS provides four standard Login modules. The standard NT and UNIX Login modules are of limited value.

JndiLoginModule
Authenticates against a directory service configured under JNDI (Java Naming and Directory Interface).

Krb5LoginModule
Authenticates using Kerberos protocols.

NTLoginModule
Authenticates using the NT security information for the current user.

UnixLoginModule
Authenticates using the UNIX security information for the current user.

The problem with using `NTLoginModule` or `UnixLoginModule` is that the telemetry (MQXR) service runs with the `mqm` identity, and not the identity of the MQTT channel. `mqm` is the identity passed to `NTLoginModule` or `UnixLoginModule` for authentication, and not the identity of the client.

To overcome this problem, write your own Login module, or use the other standard Login modules. A sample `JAASLoginModule.java` is supplied with WebSphere MQ Telemetry. It is an implementation of the `javax.security.auth.spi.LoginModule` interface. Use it to develop your own authentication method.

Any new LoginModule classes you provide must be on the class path of the telemetry (MQXR) service. Do not place your classes in WebSphere MQ directories that are in the class path. Create your own directories, and define the whole class path for the telemetry (MQXR) service.

You can augment the class path used by the telemetry (MQXR) service by setting class path in the `service.env` file. `CLASSPATH` must be capitalized, and the class path statement can only contain literals. You cannot use variables in the CLASSPATH; for example CLASSPATH=%CLASSPATH% is incorrect. The telemetry (MQXR) service sets its own classpath. The CLASSPATH defined in `service.env` is added to it.

The telemetry (MQXR) service provides two callbacks that return the `Username` and the `Password` for a client connected to the MQTT channel. The `Username` and `Password` are set in the `MqttConnectOptions` object. See Figure 30 on page 136 for an example of how to access `Username` and `Password`.

Examples

An example of a JAAS configuration file with one named configuration, `MQXRConfig`.

```
MQXRConfig {
  samples.JAASLoginModule required debug=true;
  //com.ibm.security.auth.module.NTLoginModule required;
  //com.ibm.security.auth.module.Krb5LoginModule required
  //                 principal=principal@your_realm
  //                 useDefaultCcache=TRUE
  //                 renewTGT=true;
  //com.sun.security.auth.module.NTLoginModule required;
  //com.sun.security.auth.module.UnixLoginModule required;
  //com.sun.security.auth.module.Krb5LoginModule required
  //                 useTicketCache="true"
  //                 ticketCache="${user.home}${/}tickets";
};
```

Figure 29. Sample `jaas.config` file

An example of a JAAS Login module coded to receive the `Username` and `Password` provided by an MQTT client.

```
  public boolean login()
      throws javax.security.auth.login.LoginException {
    javax.security.auth.callback.Callback[] callbacks =
      new javax.security.auth.callback.Callback[2];
    callbacks[0] = new javax.security.auth.callback.NameCallback("NameCallback");
    callbacks[1] = new javax.security.auth.callback.PasswordCallback(
        "PasswordCallback", false);
    try {
      callbackHandler.handle(callbacks);
      String username = ((javax.security.auth.callback.NameCallback) callbacks[0])
          .getName();
      char[] password = ((javax.security.auth.callback.PasswordCallback) callbacks[1])
          .getPassword();
      // Accept everything.
      if (true) {
        loggedIn = true;
      } else
        throw new javax.security.auth.login.FailedLoginException("Login failed");

      principal= new JAASPrincipal(username);

    } catch (java.io.IOException exception) {
      throw new javax.security.auth.login.LoginException(exception.toString());
    } catch (javax.security.auth.callback.UnsupportedCallbackException exception) {
      throw new javax.security.auth.login.LoginException(exception.toString());
    }

    return loggedIn;
  }
```

Figure 30. Sample `JAASLoginModule.Login()` method

IBM WebSphere MQ Telemetry daemon for devices concepts

The IBM WebSphere MQ Telemetry daemon for devices is an advanced MQTT V3 client application. Use it to store and forward messages from other MQTT clients. It connects to IBM WebSphere MQ like an MQTT client, but you can also connect other MQTT clients to it.

The daemon is a publish/subscribe broker. MQTT V3 clients connect to it to publish and subscribe to topics, using topic strings to publish, and topic filters to subscribe. The topic string is hierarchical, with topic levels divided by /. Topic filters are topic strings that can include single level + wildcards and a multilevel # wildcard as the last part of the topic string.

Note: Wildcards in the daemon follow the more restrictive rules of WebSphere Message Broker, v6. IBM WebSphere MQ is different. It supports mulitple multilevel wildcards; wildcards can stand in for any number of levels of the hierarchy, anywhere in the topic string.

Multiple MQTT v3 clients connect to the daemon using a listener port. The default listener port is modifiable. You can define multiple listener ports and allocate different namespaces to them, see "WebSphere MQ Telemetry daemon for devices listener ports" on page 144. The daemon is itself an MQTT v3 client. Configure a daemon bridge connection to connect the daemon to the listener port of another daemon, or to a WebSphere MQ Telemetry (MQXR) service.

You can configure multiple bridges for the WebSphere MQ Telemetry daemon for devices. Use the bridges to connect together a network of daemons that can exchange publications.

Each bridge can publish and subscribe to topics at its local daemon. It can also publish and subscribe to topics at another daemon, a WebSphere MQ publish/subscribe broker, or any other MQTT v3 broker it is connected to. Using a topic filter, you can select the publications to propagate from one broker to another. You can propagate publications in either direction. You can propagate publicaitons from the local daemon to each of its attached remote brokers, or from any of the attached brokers to the local daemon; see "IBM WebSphere MQ Telemetry daemon for devices bridges" on page 137.

IBM WebSphere MQ Telemetry daemon for devices bridges

An IBM WebSphere MQ Telemetry daemon for devices bridge connects two publish/subscribe brokers using the MQTT v3 protocol. The bridge propagates publications from one broker to the other, in either direction. At one end is a WebSphere MQ Telemetry daemon for devices bridge connection, and at the other might be a queue manager, or another daemon. A queue manager is connected to the bridge connection using a telemetry channel. A daemon is connected to the bridge connection using a daemon listener.

IBM WebSphere MQ Telemetry daemon for devices supports one or more simultaneous connections to other brokers. The connections from the daemon are called bridges and are defined by connection entries in the daemon configuration file. The connections to IBM WebSphere MQ are made using IBM WebSphere MQ telemetry channels, as shown in the following figure:

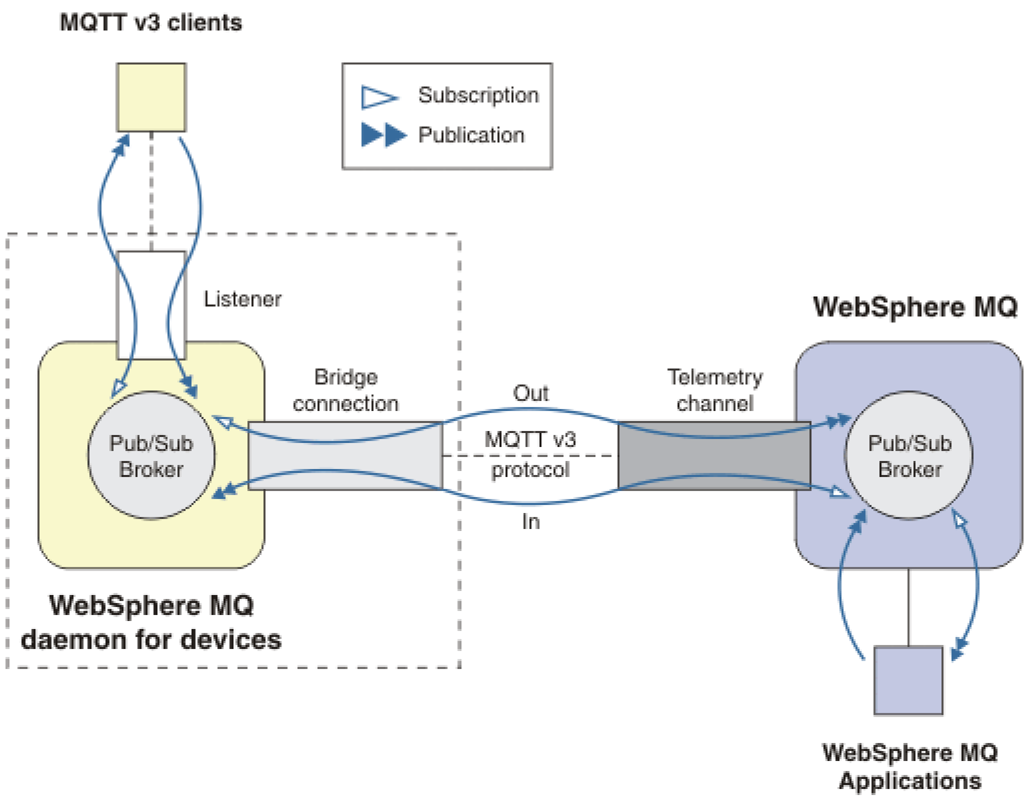


Figure 31. Connecting IBM WebSphere MQ Telemetry daemon for devices to IBM WebSphere MQ

A bridge connects the daemon to another broker as an MQTT v3 client. The bridge parameters mirror the attributes of an MQTT v3 client.

A bridge is more than a connection. It acts as a publish and subscribe agent situated between two publish/subscribe brokers. The local broker is the IBM WebSphere MQ Telemetry daemon for devices, and the remote broker is any publish/subscribe broker that supports the MQTT v3 protocol. Typically the remote broker is another daemon or IBM WebSphere MQ.

The job of the bridge is to propagate publications between the two brokers. The bridge is bidirectional. It propagates publications in either direction. Figure 31 on page 138 illustrates the way the bridge connects IBM WebSphere MQ Telemetry daemon for devices to IBM WebSphere MQ. "Example topic settings for the bridge" on page 139 uses examples to illustrate how to use the topic parameter to configure the bridge.

The In and Out arrows in Figure 31 on page 138 indicate the bidirectionality of the bridge. At one end of the arrow, a subscription is created. The publications that match the subscription are published to the broker at the opposite end of the arrow. The arrow is labeled according to the flow of publications. Publications flow In to the daemon and Out from the daemon. The importance of the labels is they are used in the command syntax. Remember that In and Out refer to where the publications flow, and not to where the subscription is sent.

Other clients, applications, or brokers might be connected either to IBM WebSphere MQ or to WebSphere MQ Telemetry daemon for devices. They publish and subscribe to topics at the broker they are connected to. If the broker is IBM WebSphere MQ, the topics might be clustered or distributed, and not explicitly defined at the local queue manager.

Uses of bridges

Connect daemons together using bridge connections and listeners. Connect daemons and queue managers together using bridge connections and telemetry channels. When you connect multiple brokers together it is possible to create loops. Be careful: Publications might circulate endlessly around a loop of brokers, undetected.

Some of the reasons for using daemons bridged to IBM WebSphere MQ are as follows:

Reduce the number of MQTT client connections to WebSphere MQ
Using a hierarchy of daemons you can connect many clients to WebSphere MQ; more clients than the number a single queue manager can connect at one time.

Store and forward messages between MQTT clients and WebSphere MQ
You might use store and forward to avoid maintaining continuous connections between clients and IBM WebSphere MQ, if the clients do not have their own storage. You might use multiple types of connections between the MQTT client and WebSphere MQ; see Telemetry concepts and scenarios for monitoring and control.

Filter the publications exchanged between MQTT clients and WebSphere MQ
Commonly, publications divide into messages that are processed locally and messages that involve other applications. Local publications might include control flows between sensors and actuators, and remote publications include requests for readings, status, and configuration commands.

Change the topic spaces of publications
Avoid topics strings from clients attached to different listener ports from colliding with one another. The example uses the daemon to label meter readings coming from different buildings; see Separating the topic spaces of different groups of clients.

Example topic settings for the bridge

Publish everything to the remote broker - using defaults

The default direction is called out, and the bridge publishes topics to the remote broker. The topic parameter controls what topics are propagated using topic filters.

The bridge uses the topic parameter in Figure 32 on page 139 to subscribe to everything published to the local daemon by MQTT clients, or by other brokers. The bridge publishes the topics to the remote broker connected by the bridge.

```
connection Daemon1
topic #
```

Figure 32. Publish everything to the remote broker

Publish everything to the remote broker - explicit

The topic setting in the following code fragment gives the same result as using the defaults. The only difference is that the **`direction`** parameter is explicit. Use the `out` direction to subscribe to the local broker, the daemon, and publish to the remote broker. Publications created on the local daemon that the bridge has subscribed to, are published at the remote broker.

```
connection Daemon1
topic # out
```

Figure 33. Publish everything to the remote broker - explicit

Publish everything to the local broker

Instead of using the direction `out`, you can set the opposite direction, `in`. The following code fragment configures the bridge to subscribe to everything published at the remote broker connected by the bridge. The bridge publishes the topics to the local broker, the daemon.

```
connection Daemon1
topic # in
```

Figure 34. Publish everything to the local broker

Publish everything from the export topic at the local broker to the import topic at the remote broker

Use two additional topic parameters, **`local_prefix`** and **`remote_prefix`**, to modify the topic filter, # in the previous examples. One parameter is used to modify the topic filter used in the subscription, and the other parameter is used to modify the topic the publication is published to. The effect is to replace the beginning of the topic string used in one broker with another topic string on the other broker.

Depending on the direction of the topic command the meaning of **`local_prefix`** and **`remote_prefix`** reverses. If the direction is `out`, the default, **`local_prefix`** is used as part of the topic subscription, and **`remote_prefix`** replaces the **`local_prefix`** part of the topic string in the remote publication. If the direction is `in`, **`remote_prefix`** becomes part of the remote subscription, and **`local_prefix`** replaces the **`remote_prefix`** part of the topic string.

The first part of a topic string is often thought of as defining a topic space. Use the additional parameters to change the topic space a topic is published to. You might do this to avoid the topic being propagated colliding with another the topic on the target broker, or to remove a mount point topic string.

As an example, in the following code fragment, all the publications to the topic string `export/#` at the daemon are republished to `import/#` at the remote broker.

```
topic # out export/ import/
```

Figure 35. Publish everything from the export topic at the local broker to the import topic at the remote broker

Publish everything to the import topic at the local broker from the export topic at the remote broker

The following code fragment shows the configuration reversed; the bridge subscribes to everything published with the `export/#` topic string at the remote broker and publishes it to `import/#` at the local broker.

```
connection Daemon1
topic # in import/ export/
```

Figure 36. Publish everything to the import topic at the local broker from the export topic at the remote broker

Publish everything from the 1884/ mount point to the remote broker with the original topic strings

In the following code fragment, the bridge subscribes to everything published by clients connected to the mount point 1884/ at the local daemon. The bridge publishes everything published to the mount point to the remote broker. The mount point string 1884/ is removed from the topics published to the remote broker. The *local_prefix* is the same as the mount point string 1884/, and the *remote_prefix* is a blank string.

```
listener 1884
mount_point 1884/
connection Daemon1
topic # out 1884/ ""
```

Figure 37. Publish everything from the 1884/ mount point to the remote broker with the original topic strings.

Separating the topic spaces of different clients connected to different daemons

An application is written for electrical power meters to publish meter readings for a building. The readings are published using MQTT clients to a daemon hosted in the same building. The topic selected for the publications is `power`. The same application is deployed to a number of buildings in a complex. For site monitoring and data storage, readings from all buildings are aggregated using bridge connections. The connections link the building daemons to WebSphere MQ at a central location.

The client applications in each building are identical, but the data must be differentiated by building. Each reading has a `power` topic and must be prefixed with the building number to distinguish it. The bridge from the first building in the complex uses the prefix `meters/building01/`, from building two the prefix is `meters/building02/`. The readings from the other buildings follow the same pattern. WebSphere MQ receives the readings with topics like `meters/building01/power`.

The example is contrived; in practice the topic space the application publishes to is likely to be configurable.

The configuration file for each daemon has a topic statement that follows the pattern in the following code fragment:

```
connection Daemon1
topic power out "" meters/building01/
```

Figure 38. Separate the topic spaces of clients connected to different daemons

Specify an empty string as a placeholder for the unused local_prefix parameter.

Separate the topic spaces of clients connected to the same daemon

Suppose that a single daemon is used to connect all the power meters. Assuming that in the application can be configured to connect to different ports, you might distinguish the buildings by attaching the meters from different buildings to different listener ports, as in the following code fragment. Again, the example is contrived; it illustrates how mount points might be used.

```
listener 1884
mount_point meters/building01/
listener 1885
mount_point meters/building02/
connection Daemon1
topic meters/+/power out
```

Figure 39. Separate the topic spaces of clients connected to the same daemon

Remap different topics for publications flowing in both directions

In the configuration in the following code fragment, the bridge subscribes to the single topic b at the remote broker and forwards publications about b to the local daemon, changing the topic to a. The bridge also subscribes to the single topic x at the local broker and forwards publications about x to the remote broker, changing the topic to `y`.

```
connection Daemon1
topic "" in a b
topic "" out x y
```

Figure 40. Remap different topics for publications flowing in both directions

An important point about this example is that different topics are subscribed to and published to at both brokers. The topics spaces at both brokers are disjoint.

Remap the same topics for publications flowing in both directions (looping)

Unlike the previous example, the configuration in Figure 41 on page 142, in general, results in a loop. In the topic statement `topic "" in a b`, the bridge subscribes to b remotely, and publishes to a locally. In the other topic statement, the bridge subscribes to a locally, and publishes to b remotely. The same configuration can be written as shown in Figure 42 on page 142.

The general result is that if a client publishes to b remotely, the publication is transferred to the local daemon as a publication on topic a. However, on being published by the bridge to the local daemon on the topic a, the publication matches the subscription made by the bridge to local topic a. The subscription is `topic "" out a b`. As a result, the publication is transferred back to the remote broker as a publication on topic b. The bridge is now subscribed to the remote topic b, and the cycle begins again.

Some brokers implement loop detection to prevent the loop happening. But the loop detection mechanism must work when different types of brokers are bridged together. Loop detection does not work if WebSphere MQ is bridged to the WebSphere MQ Telemetry daemon for devices. It does work if two IBM WebSphere MQ Telemetry daemon for devices are bridged together. By default loop detection is turned on; see try_private.

```
connection Daemon1
topic "" in a b
topic "" out a b
```

*Figure 41. **!**Remap the same topics for publications flowing in both directions*

```
connection Daemon1
topic "" both a b
```

*Figure 42. **!**Remap the same topics for publications flowing in both directions, using `both`.*

The configuration in Figure 40 on page 142 is the same as Figure 41 on page 142.

Availability of IBM WebSphere MQ Telemetry daemon for devices bridge connections

Configure multiple IBM WebSphere MQ Telemetry daemon for devices bridge connection addresses to connect to the first available remote broker. If the broker is a multi-instance queue manager, provide both of its TCP/IP addresses. Configure a primary connection to connect, or reconnect, to the primary server, when it is available.

The connection bridge parameter, addresses, is a list of TCP/IP socket addresses. The bridge attempts to connect to each address in turn, until it makes a successful connection. The round_robin and start_type connection parameters control how the addresses are used once a successful connection has been made.

If start_type is auto, manual, or lazy, then if the connection fails, the bridge attempts to reconnect. It uses each address in turn, with about a 20 second delay between each connection attempt. If `start_type` is once, then if the connection fails, the bridge does not attempt to reconnect automatically.

If round_robin is true, the bridge connection attempts start at the first address in the list and tries each address in the list in turn. It starts at the first address again, when the list is exhausted. If there is only one address in the list, it tries it again every 20 seconds.

If round_robin is false, the first address in the list, which is called the primary server, is given preference. If the first attempt to connect to the primary server fails, the bridge continues to try to reconnect to the primary server in the background. At the same time, the bridge tries to connect using the other addresses in the list. When the background attempts to connect to the primary server succeed, the bridge disconnects from the current connection, and switches to the primary server connection.

If a connection is disconnected voluntarily, for example by issuing a **`connection_stop`** command, then if the connection is restarted, it tries to use the same address again. If the connection is disconnected due to a failure to connect, or to the remote broker dropping the connection, the bridge waits 20 seconds. It then tries to connect to the next address in the list, or the same address, if there is only one address in the list.

Connecting to a multi-instance queue manager

In a multi-instance queue manager configuration, the queue manager runs on two different servers with different IP addresses. Typically telemetry channels are configured without a specific IP address. They are configured only with a port number. When the telemetry channel is started, by default it selects the first available network address on the local server.

Configure the addresses parameter of the bridge connection with the two IP addresses used by the queue manager. Set round_robin to true.

If the active queue manager instance fails, the queue manager switches over to the standby instance. The daemon detects that the connection to the active instance has broken and tries to reconnect to the standby instance. It uses the other IP address in the list of addresses configured for the bridge connection.

The queue manager to which the bridge connects is still the same queue manager. The queue manager recovers its own state. If cleansession is set to false, the bridge connection session is restored to the same state as before the failover. The connection resumes after a delay. Messages with "at least once" or "at most once" quality of service are not lost, and subscriptions continue to work.

The reconnection time depends on the number of channels and clients that restart when the standby instance starts, and how many messages were inflight. The bridge connection might try to reconnect to both IP addresses a number of times before the connection is reestablished.

Do not configure a multi-instance queue manager telemetry channel with a specific IP address. The IP address is only valid on one server.

If you are using an alternative high-availability solution, that manages the IP address, then it might be correct to configure a telemetry channel with a specific IP address.

cleansession

A bridge connection is an MQTT v3 client session. You can control whether a connection starts a new session, or whether it restores an existing session. If it restores an existing session, the bridge connection preserves the subscriptions and retained publications from the previous session.

Do not set cleansession to false if addresses lists multiple IP addresses, and the IP addresses connect to telemetry channels hosted by different queue managers, or to different telemetry daemons. Session state is not transferred between queue managers or daemons. Trying to restart an existing session on a different queue manager or daemon results in a new session being started. In-doubt messages are lost, and subscriptions might not behave as expected.

notifications

An application can keep track of whether the bridge connection is running by using notifications. A notification is a publication that has the value `1`, connected, or `0`, disconnected. It is published to *topicString* defined by the `notification_topic` parameter. The default value of *topicString* is `$SYS/broker/connection/` *clientIdentifier*`/state`. The default *topicString* contains the prefix $SYS. Subscribe to topics beginning with $SYS by defining a topic filter beginning with $SYS. The topic filter #, subscribe to everything, does not subscribe to topics beginning with $SYS on the daemon. Think of $SYS as defining a special system topic space distinct from the application topic space.

Notifications enable IBM WebSphere MQ Telemetry daemon for devices to notify MQTT clients when a bridge is connected or disconnected.

keepalive_interval

The keepalive_interval bridge connection parameter sets the interval between the bridge sending a TCP/IP ping to the remote server. The default interval is 60 seconds. The ping prevents the TCP/IP session being closed by the remote server, or by a firewall, that detects a period of inactivity on the connection.

clientid

A bridge connection is an MQTT v3 client session and has a `clientIdentifier` that is set by the bridge connection parameter clientid. If you intend reconnections to resume a previous session by setting the cleansession parameter to false, the `clientIdentifier` used in each session must be the same. The default value of clientid is *hostname.connectionName*, which remains the same.

Installation, verification, configuration, and control of the WebSphere MQ Telemetry daemon for devices

Installation, configuration, and control of the daemon is file-based.

Install the daemon by copying the Software Development Kit to the device where you are going to run the daemon .

As an example, run the MQTT client utility and connect to the WebSphere MQ Telemetry daemon for devices as the publish/subscribe broker; see Publish a message to a specific MQTT v3 client .

Configure the daemon by creating a configuration file; see WebSphere MQ Telemetry daemon for devices configuration file.

Control a running daemon by creating commands in the file, `amqtdd.upd`. Every 5 seconds the daemon reads the file, runs the commands, and deletes the file; see WebSphere MQ Telemetry daemon for devices command file.

WebSphere MQ Telemetry daemon for devices listener ports

Connect MQTT V3 clients to the WebSphere MQ Telemetry daemon for devices using listener ports. You can qualify a listener port with a mount point and a maximum number of connections.

A listener port must correspond to the port number specified on the MQTT client `connect(serverURI)` method of a client connecting to this port. It defaults on both the client and the daemon to 1883.

You can change the default port for the daemon by setting the global definition `port` in the daemon configuration file. You can set specific ports by adding a `listener` definition to the daemon configuration file.

For each listener port, other than the default port, you can specify a mount point to isolate clients. Clients connected to a port with a mount point are isolated from other clients; see "WebSphere MQ Telemetry daemon for devices mount points" on page 145.

You can limit the number of clients that can connect to any port. Set the global definition `max_connections` to limit connections to the default port, or qualify each listener port with `max_connections`.

Example

An example of a configuration file that changes the default port from 1883 to 1880, and limits connections to port 1880 to 10000. Connections to port 1884 are limited to 1000. Clients attached to port 1884 are isolated from clients attached to other ports.

```
port 1880
max_connections 10000
listener 1884
mount_point 1884/
max_connections 1000
```

WebSphere MQ Telemetry daemon for devices mount points

You can associate a mount point with a listener port used by MQTT clients to connect to a WebSphere MQ Telemetry daemon for devices. A mount point isolates the publications and subscriptions exchanged by MQTT clients using one listener port from MQTT clients connected to a different listener port.

Clients attached to a listener port with a mount point can never directly exchange topics with clients attached to any other listener ports. Clients attached to a listener port without a mount point can publish or subscribe to topics of any client. Clients are not aware of whether they are attached through a mount point or not; it makes no difference to the topics strings created by clients.

A mount point is a string of text that is prefixed to the topic string of publications and subscriptions. It is prefixed to all the topic strings created by clients attached to listener port with a mount point. The string of text is removed from all topic strings sent to clients attached to the listener port.

If a listener port has no mount point, the topic strings of publications and subscriptions created and received by clients attached to the port are not altered.

Create mount point strings with a trailing /. That way the mount point is the parent topic of the topic tree for the mount point.

Example

A configuration file contains the following listener ports:

```
listener 1883
mount_point 1883/
listener 1884 127.0.0.1
mount_point 1884/
listener 1885
```

A client, attached to port 1883, creates a subscription to MyTopic. The daemon registers the subscription as 1883/MyTopic. Another client attached to port 1883 publishes a message on the topic, MyTopic. The daemon changes the topic string to 1883/MyTopic and searches for matching subscriptions. The subscriber on port 1883 receives the publication with the original topic string MyTopic. The daemon has removed the mount point prefix from the topic string.

Another client, attached to port 1884, also publishes on the topic MyTopic. This time the daemon registers the topic as 1884/MyTopic. The subscriber on port 1883 does not receive the publication, because the different mount point results in a subscription with a different topic string.

A client, attached to port 1885, publishes on the topic, 1883/MyTopic. The daemon does not change the topic string. The subscriber on port 1883 receives the publication to MyTopic.

WebSphere MQ Telemetry daemon for devices quality of service, durable subscriptions and retained publications

Quality of service settings apply only to a running daemon. If a daemon stops, whether in a controlled manner, or because of a failure, the state of inflight messages is lost. The delivery of a message at least once, or at most once, cannot be guaranteed if the daemon stops. WebSphere MQ Telemetry daemon for devices supports limited persistence. Set the **retained_persistence** configuration parameter to save retained publications and subscriptions when the daemon is shut down.

Unlike WebSphere MQ, the WebSphere MQ Telemetry daemon for devices does not journal persistent data. Session state, message state, and retained publications are not saved transactionally. By default, the daemon discards all data when it stops. You can set an option to periodically checkpoint subscriptions and retained publications. Message status is always lost when the daemon stops. All non-retained publications are lost.

Set the daemon configuration option, `Retained_persistence` to `true`, to save retained publications periodically to a file. When the daemon restarts, the retained publications that were last autosaved are reinstated. By default, retained messages created by clients are not reinstated when the daemon restarts.

Set the daemon configuration option, `Retained_persistence` to `true`, to save subscriptions created in a persistent session periodically to a file. If `Retained_persistence` is set to `true`, subscriptions that clients create in a session with `CleanSession` set to `false`, a "persistent session", are restored. The daemon restores the subscriptions when it restarts, which start receiving publications. The client receives the publications when it restarts with `CleanSession` to `false`. By default, client session state is not saved when a daemon stops, and so subscriptions are not restored, even if the client sets `CleanSession` to `false`.

`Retained_persistence` is an autosave mechanism. It might not save the most recent retained publications or subscriptions. You can change how often retained publications and subscriptions are saved. Set the interval between saves, or the number of changes between saves, using the configuration options `autosave_on_changes` and `autosave_interval`.

Example configuration for setting persistence

```
# Sample configuration
# Daemon listens on port 1882 with persistence in /tmp
# Autosave every minute
port 1882
persistence_location /tmp/
retained_persistence true
autosave_on_changes false
autosave_interval 60
```

WebSphere MQ Telemetry daemon for devices security

The WebSphere MQ Telemetry daemon for devices can authenticate clients that connect to it, use credentials to connect to other brokers, and control access to topics. The security the daemon provides is limited by being built using the WebSphere MQ Telemetry C client, which does not provide SSL support. Consequently, connections to and from the daemon are not encrypted, and cannot be authenticated using certificates.

By default, no security is switched on.

Authentication of clients

MQTT clients can set a username and password using the methods `MqttConnectOptions.setUserName` and `MqttConnectOptions.setPassword`.

Authenticate a client that connects to the daemon by checking the username and password provided by a client against entries in the password file. To enable authentication, create a password file and set the `password_file` parameter in the daemon configuration file; see password_file.

Set the `allow_anonymous` parameter in the daemon configuration file to allow clients connecting without usernames or passwords to connect to a daemon that is checking authentication; see allow_anonymous. If a client does provide a username or password it is always checked against the password file, if the `password_file` parameter is set.

Set the `clientid_prefixes` parameter in the daemon configuration file to limit connections to specific clients. The clients must have `clientIdentifiers` that start with one of the prefixes listed in the `clientid_prefixes` parameter; see clientid_prefixes.

Bridge connection security

Each WebSphere MQ Telemetry daemon for devices bridge connection is an MQTT V3 client. You can set the username and password for each bridge connection as a bridge connection parameter in the daemon configuration file; see username and password. A bridge can then authenticate itself to a broker.

Access control of topics

If clients are being authenticated, the daemon can also provide control access to topics for each user. The daemon grants access control based on matching the topic to which a client is either publishing or subscribing with an access topic string in the access control file; see acl_file.

The access control list has two parts. The first part controls access for all clients, including anonymous clients. The second part has a section for any user in the password file. It lists specific access control for each user.

Example

The security parameters are shown in the following example.

```
acl_file c:\WMQTDaemon\config\acl.txt
password_file c:\WMQTDaemon\config\passwords.txt
allow_anonymous true
connection Daemon1
username daemon1
password deamonpassword
```

Figure 43. Daemon configuration file

```
Fred:Fredpassword
Barney:Barneypassword
```

Figure 44. Password file, passwords.txt

```
topic home/public/#
topic read meters/#
user Fred
topic write meters/fred
topic home/fred/#
user Barney
topic write meters/barney
topic home/barney/#
```

Figure 45. Access control file, acl.txt

Administering multicast

Use this information to learn about the WebSphere MQ Multicast administration tasks such as reducing the size of multicast messages and enabling data conversion.

Getting started with multicast

Use this information to get started with WebSphere MQ Multicast topics and communication information objects.

About this task

WebSphere MQ Multicast messaging uses the network to deliver messages by mapping topics to group addresses. The following tasks are a quick way to test if the required IP address and port are correctly configured for multicast messaging.

Creating a COMMINFO object for multicast

The communication information (COMMINFO) object contains the attributes associated with multicast transmission. For more information about the COMMINFO object parameters, see DEFINE COMMINFO.

Use the following command-line example to define a COMMINFO object for multicast:

```
DEFINE COMMINFO(MC1) GRPADDR(group address) PORT(port number)
```

where *MC1* is the name of your COMMINFO object, *group address* is your group multicast IP address or DNS name, and the *port number* is the port to transmit on (The default value is 1414).

A new COMMINFO object called *MC1* is created; This name is the name that you must specify when defining a TOPIC object in the next example.

Creating a TOPIC object for multicast

A topic is the subject of the information that is published in a publish/subscribe message, and a topic is defined by creating a TOPIC object. TOPIC objects have two parameters which define whether they can be used with multicast or not. These parameters are: **COMMINFO** and **MCAST**.

- **COMMINFO** This parameter specifies the name of the multicast communication information object. For more information about the COMMINFO object parameters, see DEFINE COMMINFO.
- **MCAST** This parameter specifies whether multicast is allowable at this position in the topic tree.

Use the following command-line example to define a TOPIC object for multicast:

```
DEFINE TOPIC(ALLSPORTS) TOPICSTR('Sports') COMMINFO(MC1) MCAST(ENABLED)
```

A new TOPIC object called *ALLSPORTS* is created. It has a topic string *Sports*, its related communication information object is called *MC1* (which is the name you specified when defining a COMMINFO object in the previous example), and multicast is enabled.

Testing the multicast publish/subscribe

After the TOPIC and COMMINFO objects have been created, they can be tested using the amqspubc sample and the amqssubc sample. For more information about these samples see The Publish/Subscribe sample programs.

1. Open two command-line windows; The first command line is for the amqspubc publish sample, and the second command line is for the amqssubc subscribe sample.
2. Enter the following command at command line 1:

   ```
   amqspubc Sports QM1
   ```

   where *Sports* is the topic string of the TOPIC object defined in an earlier example, and *QM1* is the name of the queue manager.
3. Enter the following command at command line 2:

```
amqssubc Sports QM1
```

where *Sports* and *QM1* are the same as used in step "2" on page 148.

4. Enter `Hello world` at command line 1. If the port and IP address that are specified in the COMMINFO object are configured correctly; the `amqssubc` sample, which is listening on the port for publications from the specified address, outputs `Hello world` at command line 2.

IBM WebSphere MQ Multicast topic topology

Use this example to understand the IBM WebSphere MQ Multicast topic topology.

IBM WebSphere MQ Multicast support requires that each subtree has its own multicast group and data stream within the total hierarchy.

The *classful network* IP addressing scheme has designated address space for multicast address. The full multicast range of IP address is `224.0.0.0` to `239.255.255.255`, but some of these addresses are reserved. For a list of reserved address either contact your system administrator or see IPv4 Multicast Address Space Registry for more information. It is recommended that you use the locally scoped multicast address in the range of `239.0.0.0` to `239.255.255.255`.

In the following diagram, there are two possible multicast data streams:

```
DEF COMMINFO(MC1) GRPADDR(239.XXX.XXX.XXX
)

DEF COMMINFO(MC2) GRPADDR(239.YYY.YYY.YYY)
```

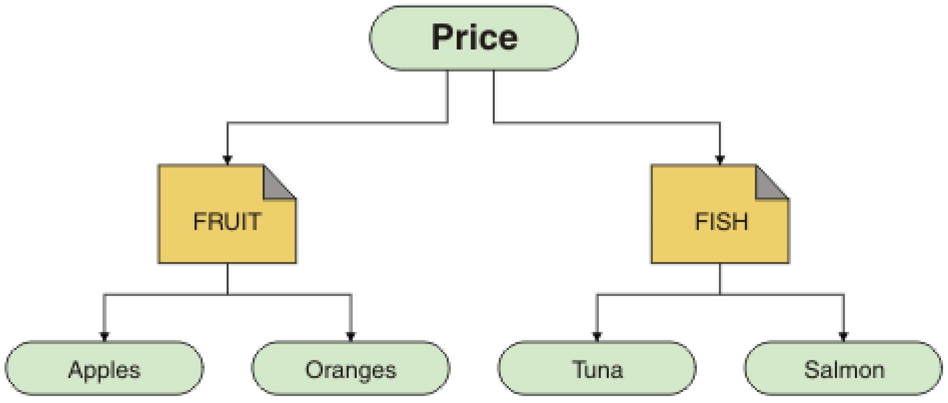
where *239.XXX.XXX.XXX* and *239.YYY.YYY.YYY* are valid multicast addresses.

These topic definitions are used to create a topic tree as shown in the following diagram:

```

DEFINE TOPIC(FRUIT) TOPICSTRING('Price/FRUIT') MCAST(ENABLED) COMMINFO(MC1)

DEFINE TOPIC(FISH) TOPICSTRING('Price/FISH') MCAST(ENABLED) COMMINFO(MC2)
```



Each multicast communication information (COMMINFO) object represents a different stream of data because their group addresses are different. In this example, the FRUIT topic is defined to use COMMINFO object MC1, the `FISH` topic is defined to use COMMINFO object `MC2`, and the `Price` node has no multicast definitions.

WebSphere MQ Multicast has a 255 character limit for topic strings. This limitation means that care must be taken with the names of nodes and leaf-nodes within the tree; if the names of nodes and leaf-nodes are too long, the topic string might exceed 255 characters and return the 2425 (0979) (RC2425): MQRC_TOPIC_STRING_ERROR reason code. It is recommended to make topic strings as short as possible because longer topic strings might have a detrimental effect on performance.

Controlling the size of multicast messages

Use this information to learn about the WebSphere MQ message format, and reduce the size of WebSphere MQ messages.

WebSphere MQ messages have a number of attributes associated with them which are contained in the message descriptor. For small messages, these attributes might represent most of the data traffic and can have a significant detrimental effect on the transmission rate. WebSphere MQ Multicast enables the user to configure which, if any, of these attributes are transmitted along with the message.

The presence of message attributes, other than topic string, depends on whether the COMMINFO object states that they must be sent or not. If an attribute is not transmitted, the receiving application applies a default value. The default MQMD values are not necessarily the same as the MQMD_DEFAULT value, and are described in Table 9 on page 150.

The COMMINFO object contains the `MCPROP` attribute which controls how many of the MQMD fields and user properties flow with the message. By setting the value of this attribute to an appropriate level, you can control the size of the WebSphere MQ Multicast messages:

MCPROP
The multicast properties control how many of the MQMD properties and user properties flow with the message.

ALL
All user properties and all the fields of the MQMD are transmitted.

REPLY
Only user properties, and MQMD fields that deal with replying to the messages, are transmitted. These properties are:

- MsgType
- MessageId
- CorrelId
- ReplyToQ
- ReplyToQmgr

USER
Only the user properties are transmitted.

NONE
No user properties or MQMD fields are transmitted.

COMPAT
This value causes the transmission of the message to be done in a compatible mode to RMM, which allows some inter-operation with the current XMS applications and WebSphere Message Broker RMM applications.

Multicast message attributes

Message attributes can come from various places, such as the MQMD, the fields in the MQRFH2, and message properties.

The following table shows what happens when messages are sent subject to the value of MCPROP and the default value used when an attribute is not sent.

Table 9. Messaging attributes and how they relate to multicast

Attribute	Action when using multicast	Default if not transmitted
TopicString	Always Included	Not applicable
MQMQ StrucId	Not transmitted	Not applicable
MQMD Version	Not transmitted	Not applicable

Table 9. Messaging attributes and how they relate to multicast (continued)

Attribute	Action when using multicast	Default if not transmitted
Report	Included if not default	0
MsgType	Included if not default	MQMT_DATAGRAM
Expiry	Included if not default	0
Feedback	Included if not default	0
Encoding	Included if not default	MQENC_NORMAL(equiv)
CodedCharSetId	Included if not default	1208
Format	Included if not default	MQRFH2
Priority	Included if not default	4
Persistence	Included if not default	MQPER_NOT_PERSISTENT
MsgId	Included if not default	Null
CorrelId	Included if not default	Null
BackoutCount	Included if not default	0
ReplyToQ	Included if not default	Blank
ReplyToQMgr	Included if not default	Blank
UserIdentifier	Included if not default	Blank
AccountingToken	Included if not default	Null
PutApplType	Included if not default	MQAT_JAVA
PutApplName	Included if not default	Blank
PutDate	Included if not default	Blank
PutTime	Included if not default	Blank
ApplOriginData	Included if not default	Blank
GroupID	Excluded	Not applicable
MsgSeqNumber	Excluded	Not applicable
Offset	Excluded	Not applicable
MsgFlags	Excluded	Not applicable
OriginalLength	Excluded	Not applicable
UserProperties	Included	Not applicable

Related reference
ALTER COMMINFO
DEFINE COMMINFO

Enabling data conversion for Multicast messaging

Use this information to understand how data conversion works for WebSphere MQ Multicast messaging.

WebSphere 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 WebSphere MQ data conversion is required, the conversion is performed locally at each client.

In a mixed platform installation, it might be that most of the clients require the data in a format that is not the native format of the transmitting application. In this situation the **CCSID** and **ENCODING** values of the multicast COMMINFO object can be used to define the encoding of the message transmission for efficiency.

WebSphere MQ Multicast supports data conversion of the message payload for the following built in formats:

- MQADMIN
- MQEVENT
- MQPCF
- MQRFH
- MQRFH2
- MQSTR

In addition to these formats, you can also define your own formats and use an MQDXP - Data-conversion exit parameter data conversion exit.

For information about programming data conversions, see Data conversion in the MQI 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 application monitoring

Use this information to learn about administering and monitoring WebSphere MQ Multicast.

The status of the current publishers and subscribers for multicast traffic (for example, the number of messages sent and received, or the number of messages lost) is periodically transmitted to the server from the client. When status is received, the COMMEV attribute of the COMMINFO object specifies whether or not the queue manager puts an event message on the SYSTEM.ADMIN.PUBSUB.EVENT. The event message contains the status information received. This information is an invaluable diagnostic aid in finding the source of a problem.

Use the MQSC command **DISPLAY CONN** to display connection information about the applications connected to the queue manager. For more information on the **DISPLAY CONN** command, see DISPLAY CONN.

Use the MQSC command **DISPLAY TPSTATUS** to display the status of your publishers and subscribers. For more information on the **DISPLAY TPSTATUS** command, see DISPLAY TPSTATUS.

COMMEV and the multicast message reliability indicator

The *reliability indicator*, used in conjunction with the **COMMEV** attribute of the COMMINFO object, is a key element in the monitoring of WebSphere MQ Multicast publishers and subscribers. The reliability indicator (the **MSGREL** field that is returned on the Publish or Subscribe status commands) is a WebSphere MQ indicator that illustrates the percentage of transmissions that have no errors Sometimes messages have to be retransmitted due to a transmission error, which is reflected in the value of **MSGREL**. Potential causes of transmission errors include slow subscribers, busy networks, and network outages. **COMMEV** controls whether event messages are generated for multicast handles that are created using the COMMINFO object and is set to one of three possible values:

DISABLED
Event messages are not written.

ENABLED
Event messages are always written, with a frequency defined in the COMMINFO **MONINT** parameter.

EXCEPTION

Event messages are written if the message reliability is below the reliability threshold. A message reliability level of 90% or less indicates that there might be a problem with the network configuration, or that one or more of the Publish/Subscribe applications is running too slowly:

- A value of **`MSGREL(100,100)`** indicates that there have been no issues in either the short term, or the long-term time frame.
- A value of **`MSGREL(80,60)`** indicates that 20% of the messages are currently having issues, but that it is also an improvement on the long-term value of 60.

Clients might continue transmitting and receiving multicast traffic even when the unicast connection to the queue manager is broken, therefore the data might be out of date.

Multicast message reliability

Use this information to learn how to set the WebSphere MQ Multicast subscription and message history.

A key element of overcoming transmission failure with multicast is WebSphere MQ's buffering of transmitted data (a history of messages to be kept at the transmitting end of the link). This process means that no buffering of messages is required in the putting application process because WebSphere MQ provides the reliability. The size of this history is configured via the communication information (COMMINFO) object, as described in the following information. A bigger transmission buffer means that there is more transmission history to be retransmitted if needed, but due to the nature of multicast, 100% assured delivery cannot be supported.

The WebSphere MQ Multicast message history is controlled in the communication information (COMMINFO) object by the **`MSGHIST`** attribute:

MSGHIST

This value is the amount of message history in kilobytes that is kept by the system to handle retransmissions in the case of NACKs (negative acknowledgments).

A value of `0` gives the least level of reliability. The default value is `100` KB.

The WebSphere MQ Multicast new subscription history is controlled in the communication information (COMMINFO) object by the **`NSUBHIST`** attribute:

NSUBHIST

The new subscriber history controls whether a subscriber joining a publication stream receives as much data as is currently available, or receives only publications made from the time of the subscription.

NONE

A value of `NONE` causes the transmitter to transmit only publication made from the time of the subscription. `NONE` is the default value.

ALL

A value of `ALL` causes the transmitter to retransmit as much history of the topic as is known. In some circumstances, this situation can give a similar behavior to retained publications.

Note: Using the value of `ALL` might have a detrimental effect on performance if there is a large topic history because all the topic history is retransmitted.

Related reference

DEFINE COMMINFO

ALTER COMMINFO

Advanced multicast tasks

Use this information to learn about advanced WebSphere MQ Multicast administration tasks such as configuring `.ini` files and interoperability with WebSphere MQ LLM.

For considerations for security in a Multicast installation, see Multicast security .

Bridging between multicast and non-multicast publish/subscribe domains

Use this information to understand what happens when a non-multicast publisher publishes to a WebSphere MQ Multicast enabled topic.

If a non-multicast publisher publishes to a topic that is defined as **MCAST** enabled and **BRIDGE** enabled, the queue manager transmits the message out over multicast directly to any subscribers that might be listening. A multicast publisher cannot publish to topics that are not multicast enabled.

Existing topics can be multicast enabled by setting the **MCAST** and **COMMINFO** parameters of a topic object. See Initial multicast concepts for more information about these parameters.

The COMMINFO object **BRIDGE** attribute controls publications from applications that are not using multicast. If **BRIDGE** is set to ENABLED and the **MCAST** parameter of the topic is also set to ENABLED, publications from applications that are not using multicast are bridged to applications that do. For more information on the **BRIDGE** parameter, see DEFINE COMMINFO.

Configuring the .ini files for Multicast

Use this information to understand the WebSphere MQ Multicast fields in the `.ini` files.

Additional WebSphere MQ Multicast configuration can be made in an `ini` file. The specific `ini` file that you must use is dependent on the type of applications:

- Client: Configure the *MQ_DATA_PATH*/`mqclient.ini` file.
- Queue manager: Configure the *MQ_DATA_PATH*/`qmgrs`/*QMNAME*/`qm.ini` file.

where *MQ_DATA_PATH* is the location of the WebSphere MQ data directory (`/var/mqm/mqclient.ini`), and *QMNAME* is the name of the queue manager to which the `.ini` file applies.

The `.ini` file contains fields used to fine-tune the behavior of WebSphere MQ Multicast:

```
Multicast:
  Protocol              = IP | UDP
  IPVersion             = IPV4 | IPV6 | ANY | BOTH
  LimitTransRate        = DISABLED | STATIC | DYNAMIC
  TransRateLimit        = 100000
  SocketTTL             = 1
  Batch                 = NO
  Loop                  = 1
  Interface             = <IPaddress>
  FeedbackMode          = ACK | NACK | WAIT1
  HeartbeatTimeout      = 20000
  HeartbeatInterval     = 2000
```

Protocol

UDP
In this mode, packets are sent using the UDP protocol. Network elements cannot provide assistance in the multicast distribution as they do in IP mode however. The packet format remains compatible with PGM. This is the default value.

IP
In this mode, the transmitter sends raw IP packets. Network elements with PGM support assist in the reliable multicast packet distribution. This mode is fully compatible with the PGM standard.

IPVersion

IPV4
Communicate using the IPv4 protocol only. This is the default value.

IPV6
Communicate using the IPv6 protocol only.

ANY
Communicate using IPv4, IPv6, or both, depending on which protocol is available.

BOTH
Supports communication using both IPv4 and IPv6.

LimitTransRate

DISABLED
: There is no transmission rate control. This is the default value.

STATIC
: Implements static transmission rate control. The transmitter would not transmit at a rate exceeding the rate specified by the TransRateLimit parameter.

DYNAMIC
: The transmitter adapts its transmission rate according to the feedback it gets from the receivers. In this case the transmission rate limit cannot be more than the value specified by the TransRateLimit parameter. The transmitter tries to reach an optimal transmission rate.

TransRateLimit
The transmission rate limit in Kbps.

SocketTTL
The value of SocketTTL determines if the multicast traffic can pass through a router, or the number of routers it can pass through.

Batch
Controls whether messages are batched or sent immediately There are 2 possible values:

- *NO* The messages are not batched, they are sent immediately.
- *YES* The messages are batched.

Loop
Set the value to 1 to enable multicast loop. Multicast loop defines whether the data sent is looped back to the host or not.

Interface
The IP address of the interface on which multicast traffic flows. For more information and troubleshooting, see: Testing multicast applications on a non-multicast network and Setting the appropriate network for multicast traffic

FeedbackMode

NACK
: Feedback by negative acknowledgments. This is the default value.

ACK
: Feedback by positive acknowledgments.

WAIT1
: Feedback by positive acknowledgments where the transmitter waits for only 1 ACK from any of the receivers.

HeartbeatTimeout
The heartbeat timeout in milliseconds. A value of 0 indicates that the heartbeat timeout events are not raised by the receiver or receivers of the topic. The default value is 20000.

HeartbeatInterval
The heartbeat interval in milliseconds. A value of 0 indicates that no heartbeats are sent. The heartbeat interval must be considerably smaller than the **`HeartbeatTimeout`** value to avoid false heartbeat timeout events. The default value is 2000.

Multicast interoperability with WebSphere MQ Low Latency Messaging

Use this information to understand the interoperability between WebSphere MQ Multicast and WebSphere MQ Low Latency Messaging (LLM).

Basic payload transfer is possible for an application using LLM, with another application using multicast to exchange messages in both directions. Although multicast uses LLM technology, the LLM product itself is not embedded. Therefore it is possible to install both LLM and WebSphere MQ Multicast, and operate and service the two products separately.

LLM applications that communicate with multicast might need to send and receive message properties. The WebSphere MQ message properties and MQMD fields are transmitted as LLM message properties with specific LLM message property codes as shown in the following table:

Table 10. WebSphere MQ message properties to WebSphere MQ LLM property mappings

WebSphere MQ property	WebSphere MQ LLM property type	LLM property kind	LLM property code
MQMD.Report	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1001
MQMD.MsgType	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1002
MQMD.Expiry	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1003
MQMD.Feedback	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1004
MQMD.Encoding	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1005
MQMD.CodedCharSetId	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1006
MQMD.Format	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_String	-1007
MQMD.Priority	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1008
MQMD.Persistence	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1009
MQMD.MsgId	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_ByteArray	-1010
MQMD.BackoutCount	RMM_MSG_PROP_INT32	LLM_PROP_KIND_Int32	-1012
MQMD.ReplyToQ	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_String	-1013
MQMD.ReplyToQMger	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_String	-1014
MQMD.PutDate	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_String	-1020
MQMD.PutTime	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_String	-1021
MQMD.ApplOriginData	RMM_MSG_PROP_BYTES	LLM_PROP_KIND_String	-1022
MQPubOptions	RMM_MSG_PROP_INT32	LLM_PROP_KIND_int32	-1053

For more information about LLM, see the LLM product documentation: WebSphere MQ Low Latency Messaging.

Administering HP Integrity NonStop Server

Use this information to learn about administration tasks for the IBM WebSphere MQ client for HP Integrity NonStop Server.

Two administration tasks are available to you:

1. Manually starting the TMF/Gateway from Pathway.
2. Stopping the TMF/Gateway from Pathway.

Manually starting the TMF/Gateway from Pathway

You can allow Pathway to automatically start the TMF/Gateway on the first enlistment request, or you can manually start the TMF/Gateway from Pathway.

Procedure

To manually start the TMF/Gateway from Pathway, enter the following PATHCOM command:

```
START SERVER <server_class_name>
```

If a client application makes an enlistment request before the TMF/Gateway completes recovery of in-doubt transactions, the request is held for up to 1 second. If recovery does not complete within that time, the enlistment is rejected. The client then receives an MQRC_UOW_ENLISTMENT_ERROR error from use of a transactional MQI.

Stopping the TMF/Gateway from Pathway

This task describes how to stop the TMF/Gateway from Pathway, and how to restart the TMF/Gateway after you stop it.

Procedure

1. To prevent any new enlistment requests being made to the TMF/Gateway, enter the following command:

   ```
   FREEZE SERVER <server_class_name>
   ```

2. To trigger the TMF/Gateway to complete any in-flight operations and to end, enter the following command:

   ```
   STOP SERVER <server_class_name>
   ```

3. To allow the TMF/Gateway to restart either automatically on first enlistment or manually, following steps 1 and 2, enter the following command:

   ```
   THAW SERVER <server_class_name>
   ```

   Applications are prevented from making new enlistment requests and it is not possible to issue the **START** command until you issue the **THAW** command.

Notices

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 WebSphere 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

Part Number:

(1P) P/N: