7.5

Monitoring and Performance for IBM WebSphere MQ

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 283.

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.

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

Contents

Monitoring and performance

A number of monitoring techniques are available in IBM WebSphere MQ to obtain statistics and other specific information about how your queue manager network is running. Use the monitoring information and guidance in this section to help improve the performance of your queue manager network.

Depending on the size and complexity of your queue manager network, you can obtain a range of information from monitoring your queue manager network. The following list provides examples of reasons for monitoring your queue manager network:

- Detect problems in your queue manager network.
- Assist in determining the causes of problems in your queue manager network.
- Improve the efficiency of your queue manager network.
- Familiarize yourself with the running of your queue manager network.
- Confirm that your queue manager network is running correctly.
- Generate messages when certain events occur.
- Record message activity.
- Determine the last known location of a message.
- Check various statistics of a queue manager network in real time.
- Generate an audit trail.
- Account for application resource usage.
- Capacity planning.

Related tasks
Configuring
Administering WebSphere MQ

Event monitoring

Event monitoring is the process of detecting occurrences of *instrumentation events* in a queue manager network. An instrumentation event is a logical combination of events that is detected by a queue manager or channel instance. Such an event causes the queue manager or channel instance to put a special message, called an *event message*, on an event queue.

IBM WebSphere MQ instrumentation events provide information about errors, warnings, and other significant occurrences in a queue manager. Use these events to monitor the operation of the queue managers in your queue manager network to achieve the following goals:

- Detect problems in your queue manager network.
- Assist in determining the causes of problems in your queue manager network.
- Generate an audit trail.
- React to queue manager state changes

Related reference
Event message reference
"Event types" on page 8
Use this page to view the types of instrumentation event that a queue manager or channel instance can report

Event message format

Instrumentation events

An instrumentation event is a logical combination of conditions that a queue manager or channel instance detects and puts a special message, called an *event message*, on an event queue.

IBM WebSphere MQ instrumentation events provide information about errors, warnings, and other significant occurrences in a queue manager. You can use these events to monitor the operation of queue managers (with other methods such as Tivoli® NetView for z/OS®).

illustrates the concept of instrumentation events.

1. Event conditions

2. Event message put on event queue

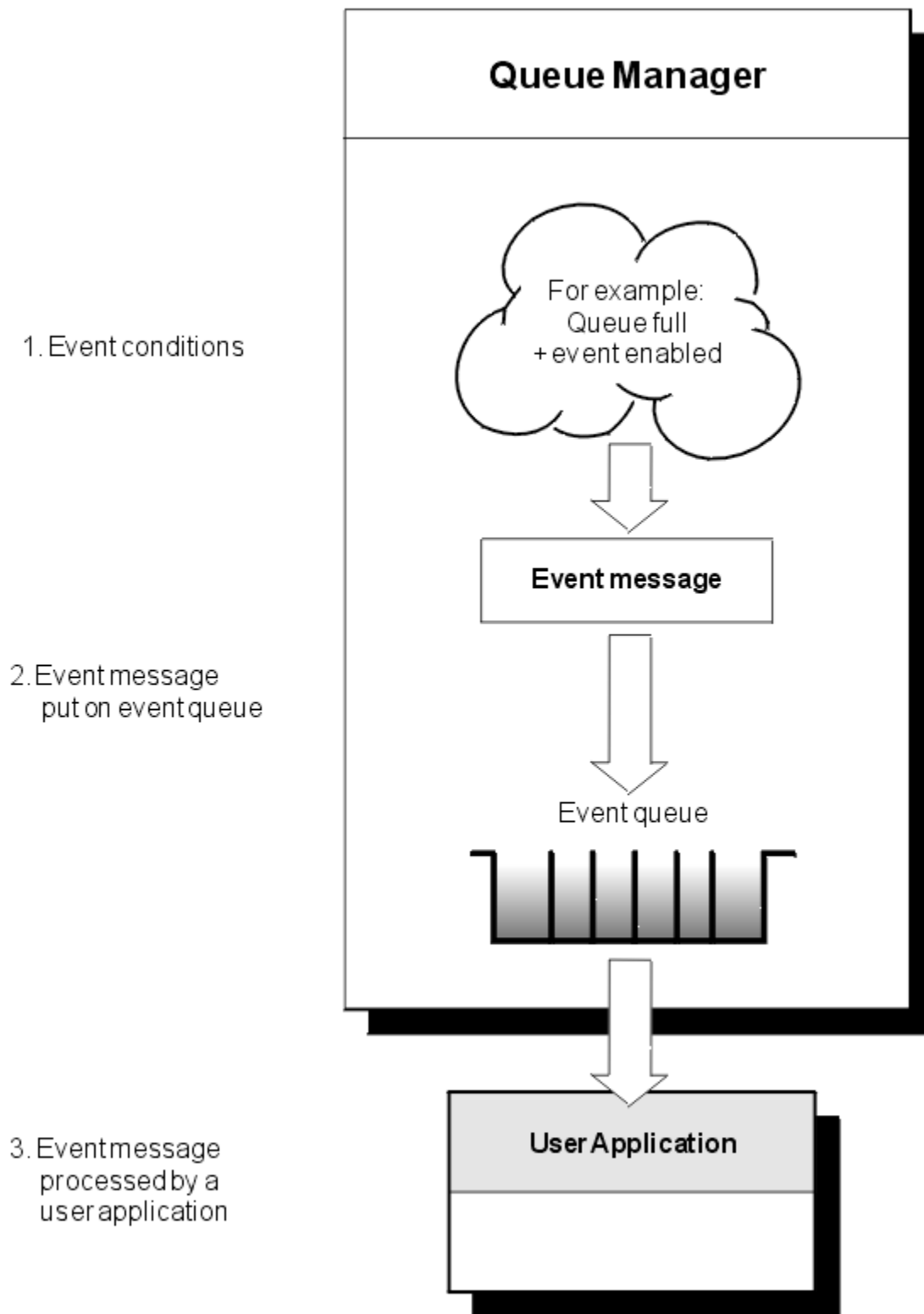3. Event message processed by a user application

Figure 1. Understanding instrumentation events

Event monitoring applications

Applications that use events to monitor queue managers must include the following provisions:

1. Set up channels between the queue managers in your network.

2. Implement the required data conversions. The normal rules of data conversion apply. For example, if you are monitoring events on a UNIX® system queue manager from a z/OS queue manager, ensure that you convert EBCDIC to ASCII.

Event notification through event queues

When an event occurs, the queue manager puts an event message on the appropriate event queue, if defined. The event message contains information about the event that you can retrieve by writing a suitable MQI application program that performs the following steps:

- Get the message from the queue.
- Process the message to extract the event data.

The related information describes the format of event messages.

Conditions that cause events

The following list gives examples of conditions that can cause instrumentation events:

- A threshold limit for the number of messages on a queue is reached.
- A channel instance is started or stopped.
- A queue manager becomes active, or is requested to stop.
- An application tries to open a queue specifying a user ID that is not authorized on IBM WebSphere MQ for IBM i, Windows, UNIX and Linux® systems.
- Objects are created, deleted, changed, or refreshed.
- An MQSC or PCF command runs successfully.
- A queue manager starts writing to a new log extent.
- Putting a message on the dead-letter queue, if the event conditions are met.

Related concepts
"Performance events" on page 19
Performance events relate to conditions that can affect the performance of applications that use a specified queue. The scope of performance events is the queue. **MQPUT** calls and **MQGET** calls on one queue do not affect the generation of performance events on another queue.

"Sample program to monitor instrumentation events" on page 48
Use this page to view a sample C program for monitoring instrumentation events

Event types

Use this page to view the types of instrumentation event that a queue manager or channel instance can report

IBM WebSphere MQ instrumentation events have the following types:

- Queue manager events
- Channel and bridge events
- Performance events
- Configuration events
- Command events
- Logger events
- Local events

For each queue manager, each category of event has its own event queue. All events in that category result in an event message being put onto the same queue.

This event queue:	Contains messages from:
SYSTEM.ADMIN.QMGR.EVENT	Queue manager events
SYSTEM.ADMIN.CHANNEL.EVENT	Channel events
SYSTEM.ADMIN.PERFM.EVENT	Performance events
SYSTEM.ADMIN.CONFIG.EVENT	Configuration events
SYSTEM.ADMIN.COMMAND.EVENT	Command events
SYSTEM.ADMIN.LOGGER.EVENT	Logger events
SYSTEM.ADMIN.PUBSUB.EVENT	Gets events related to Publish/Subscribe. Only used with Multicast. For more information see, Multicast application monitoring.

By incorporating instrumentation events into your own system management application, you can monitor the activities across many queue managers, across many different nodes, and for multiple IBM WebSphere MQ applications. In particular, you can monitor all the nodes in your system from a single node (for those nodes that support IBM WebSphere MQ events) as shown in Figure 2 on page 9.

Instrumentation events can be reported through a user-written reporting mechanism to an administration application that can present the events to an operator.
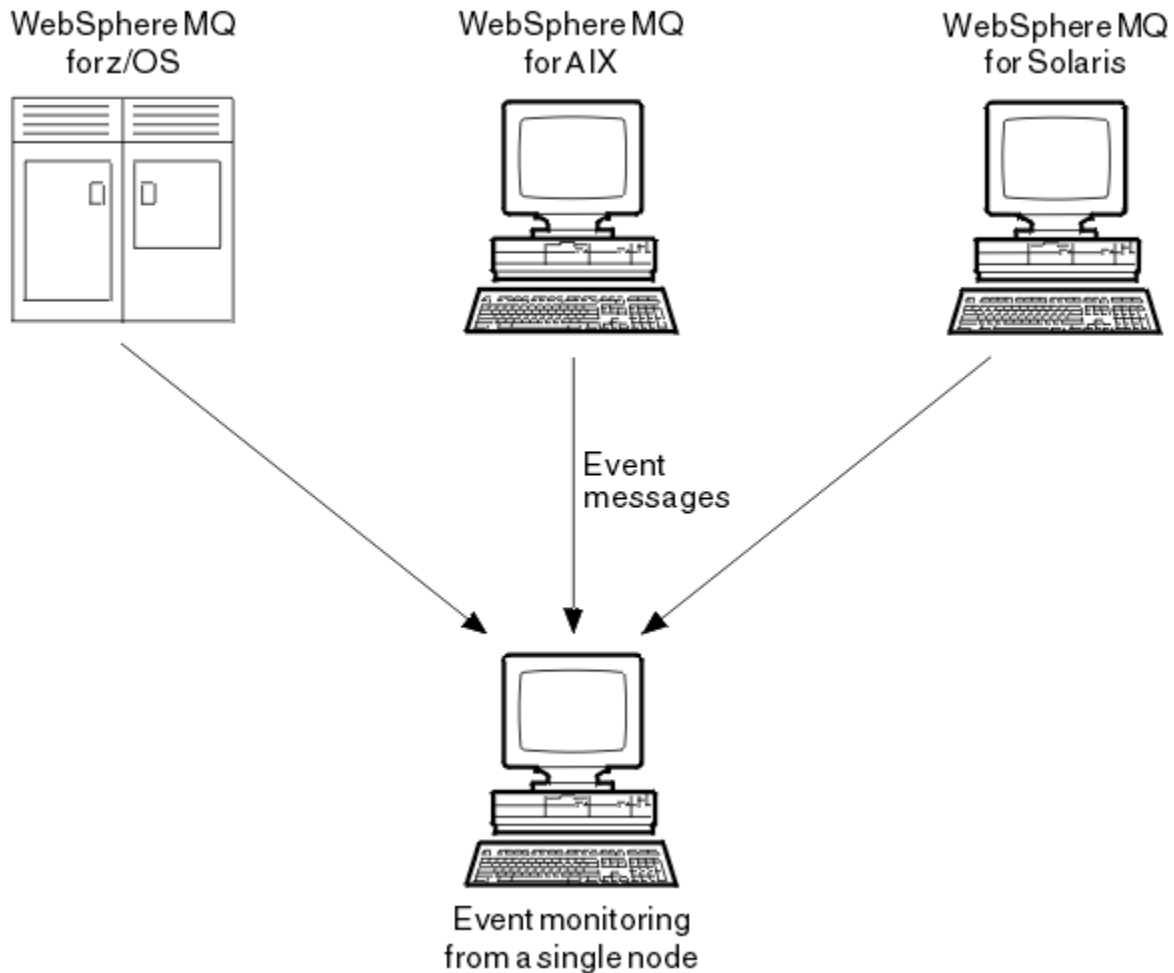

Figure 2. Monitoring queue managers across different platforms, on a single node

Instrumentation events also enable applications acting as agents for other administration networks, for example Tivoli NetView for z/OS, to monitor reports and create the appropriate alerts.

Queue manager events

Queue manager events are related to the use of resources within queue managers. For example, a queue manager event is generated if an application tries to put a message on a queue that does not exist.

The following examples are conditions that can cause a queue manager event:

- An application issues an MQI call that fails. The reason code from the call is the same as the reason code in the event message.

  A similar condition can occur during the internal operation of a queue manager; for example, when generating a report message. The reason code in an event message might match an MQI reason code, even though it is not associated with any application. Do not assume that, because an event message reason code looks like an MQI reason code, the event was necessarily caused by an unsuccessful MQI call from an application.

- A command is issued to a queue manager and processing this command causes an event. For example:

  - A queue manager is stopped or started.

  - A command is issued where the associated user ID is not authorized for that command.

WebSphere MQ puts messages for queue manager events on the SYSTEM.ADMIN.QMGR.EVENT queue, and supports the following queue manager event types:

Authority (on Windows™, and UNIX systems only)
　Authority events report an authorization, such as an application trying to open a queue for which it does not have the required authority, or a command being issued from a user ID that does not have the required authority. The authority event message can contain the following event data:

  - Not Authorized (type 1)
  - Not Authorized (type 2)
  - Not Authorized (type 3)
  - Not Authorized (type 4)
  - Not Authorized (type 5)
  - Not Authorized (type 6)

　All authority events are valid on Windows, and UNIX systems only.

Inhibit
　Inhibit events indicate that an MQPUT or MQGET operation has been attempted against a queue where the queue is inhibited for puts or gets, or against a topic where the topic is inhibited for publishes. The inhibit event message can contain the following event data:

  - Get Inhibited
  - Put Inhibited

Local
　Local events indicate that an application (or the queue manager) has not been able to access a local queue or other local object. For example, an application might try to access an object that has not been defined. The local event message can contain the following event data:

  - Alias Base Queue Type Error
  - Unknown Alias Base Queue
  - Unknown Object Name

Remote
　Remote events indicate that an application or the queue manager cannot access a remote queue on another queue manager. For example, the transmission queue to be used might not be correctly defined. The remote event message can contain the following event data:

  - Default Transmission Queue Type Error
  - Default Transmission Queue Usage Error

- Queue Type Error
- Remote Queue Name Error
- Transmission Queue Type Error
- Transmission Queue Usage Error
- Unknown Default Transmission Queue
- Unknown Remote Queue Manager
- Unknown Transmission Queue

Start and stop

Start and stop events indicate that a queue manager has been started or has been requested to stop or quiesce.

z/OS supports only start events.

Stop events are not recorded unless the default message-persistence of the SYSTEM.ADMIN.QMGR.EVENT queue is defined as persistent. The start and stop event message can contain the following event data:

- Queue Manager Active
- Queue Manager Not Active

For each event type in this list, you can set a queue manager attribute to enable or disable the event type.

Channel and bridge events

Channels report these events as a result of conditions detected during their operation. For example, when a channel instance is stopped.

Channel events are generated in the following circumstances:

- When a command starts or stops a channel.
- When a channel instance starts or stops.
- When a channel receives a conversion error warning when getting a message.
- When an attempt is made to create a channel automatically; the event is generated whether the attempt succeeds or fails.

Note: Client connections do not cause Channel Started or Channel Stopped events.

When a command is used to start a channel, an event is generated. Another event is generated when the channel instance starts. However, starting a channel by a listener, the **runmqchl** command, or a queue manager trigger message does not generate an event. In these cases, an event is generated only when the channel instance starts.

A successful start or stop channel command generates at least two events. These events are generated for both queue managers connected by the channel (providing they support events).

If a channel event is put on an event queue, an error condition causes the queue manager to create an event.

The event messages for channel and bridge events are put on the SYSTEM.ADMIN.CHANNEL.EVENT queue.

The channel event messages can contain the following event data:

- Channel Activated
- Channel Auto-definition Error
- Channel Auto-definition OK
- Channel Conversion Error
- Channel Not Activated
- Channel Started

- Channel Stopped
- Channel Stopped By User
- Channel Blocked

SSL events

The only Secure Sockets Layer (SSL or TLS) event is the Channel SSL Error event. This event is reported when a channel using SSL or TLS fails to establish an SSL connection.

The SSL event messages can contain the following event data:

- Channel SSL Error
- Channel SSL Warning

Performance events

Performance events are notifications that a resource has reached a threshold condition. For example, a queue depth limit has been reached.

Performance events relate to conditions that can affect the performance of applications that use a specified queue. They are not generated for the event queues themselves.

The event type is returned in the command identifier field in the message data.

If a queue manager tries to put a queue manager event or performance event message on an event queue and an error that would typically create an event is detected, another event is not created and no action is taken.

MQGET and MQPUT calls within a unit of work can generate performance events regardless of whether the unit of work is committed or backed out.

The event messages for performance events are put on the SYSTEM.ADMIN.PERFM.EVENT queue.

There are two types of performance event:

Queue depth events
Queue depth events relate to the number of messages on a queue; that is, how full or empty the queue is. These events are supported for shared queues. The queue depth event messages can contain the following event data:

- Queue Depth High
- Queue Depth Low
- Queue Full

Queue service interval events
Queue service interval events relate to whether messages are processed within a user-specified time interval. These events are not supported for shared queues.

Configuration events

Configuration events are generated when a configuration event is requested explicitly, or automatically when an object is created, modified, or deleted.

A configuration event message contains information about the attributes of an object. For example, a configuration event message is generated if a namelist object is created, and contains information about the attributes of the namelist object.

The event messages for configuration events are put on the SYSTEM.ADMIN.CONFIG.EVENT queue.

There are four types of configuration event:

Create object events
Create object events are generated when an object is created. The event message contains the following event data: Create object .

Change object events
> Change object events are generated when an object is changed. The event message contains the following event data: Change object .

Delete object events
> Delete object events are generated when an object is deleted. The event message contains the following event data: Delete object .

Refresh object events
> Refresh object events are generated by an explicit request to refresh. The event message contains the following event data: Refresh object .

Command events

Command events are reported when an MQSC or PCF command runs successfully.

A command event message contains information about the origin, context, and content of a command. For example, a command event message is generated with such information if the MQSC command, ALTER QLOCAL, runs successfully.

The event messages for command events are put on the SYSTEM.ADMIN.COMMAND.EVENT queue.

Command events contain the following event data: Command .

Logger events

Logger events are reported when a queue manager that uses linear logging starts writing log records to a new log extent.

A logger event message contains information specifying the log extents required by the queue manager to restart the queue manager, or for media recovery.

The event messages for logger events are put on the SYSTEM.ADMIN.LOGGER.EVENT queue.

The logger event message contains the following event data: Logger .

Event message data summary

Use this summary to obtain information about the event data that each type of event message can contain.

Event type	See these topics
Authority events	Not Authorized (type 1)
	Not Authorized (type 2)
	Not Authorized (type 3)
	Not Authorized (type 4)
	Not Authorized (type 5)
	Not Authorized (type 6)

Event type	See these topics
Channel events	Channel Activated
	Channel Auto-definition Error
	Channel Auto-definition OK
	Channel Blocked
	Channel Conversion Error
	Channel Not Activated
	Channel Started
	Channel Stopped
	Channel Stopped By User
Command events	Command
Configuration events	Create object
	Change object
	Delete object
	Refresh object
IMS Bridge events	Bridge Started
	Bridge Stopped
Inhibit events	Get Inhibited
	Put Inhibited
Local events	Alias Base Queue Type Error
	Unknown Alias Base Queue
	Unknown Object Name
Logger events	Logger
Performance events	Queue Depth High
	Queue Depth Low
	Queue Full
	Queue Service Interval High
	Queue Service Interval OK

Event type	See these topics
Remote events	Default Transmission Queue Type Error
	Default Transmission Queue Usage Error
	Queue Type Error
	Remote Queue Name Error
	Transmission Queue Type Error
	Transmission Queue Usage Error
	Unknown Default Transmission Queue
	Unknown Remote Queue Manager
	Unknown Transmission Queue
SSL events	Channel SSL Error
Start and stop events	Queue Manager Active
	Queue Manager Not Active

Controlling events

You enable and disable events by specifying the appropriate values for queue manager, queue attributes, or both, depending on the type of event.

You must enable each instrumentation event that you want to be generated. For example, the conditions causing a Queue Full event are:

- Queue Full events are enabled for a specified queue, and
- An application issues an MQPUT request to put a message on that queue, but the request fails because the queue is full.

Enable and disable events by using any of the following techniques:

- IBM WebSphere MQ script commands (MQSC).
- The corresponding IBM WebSphere MQ PCF commands.
- The IBM WebSphere MQ Explorer.

Note: You can set attributes related to events for both queues and queue managers only by command. The MQI call MQSET does not support attributes related to events.

Related concepts
"Instrumentation events" on page 6
An instrumentation event is a logical combination of conditions that a queue manager or channel instance detects and puts a special message, called an *event message*, on an event queue.

Related tasks
Automating administration tasks
Using Programmable Command Formats
Related reference
"Event types" on page 8
Use this page to view the types of instrumentation event that a queue manager or channel instance can report

The MQSC commands

Controlling queue manager events

You control queue manager events by using queue manager attributes. To enable queue manager events, set the appropriate queue manager attribute to ENABLED. To disable queue manager events, set the appropriate queue manager attribute to DISABLED.

To enable or disable queue manager events, use the MQSC command ALTER QMGR, specifying the appropriate queue manager attribute. summarizes how to enable queue manager events. To disable a queue manager event, set the appropriate parameter to DISABLED.

Table 1. Enabling queue manager events using MQSC commands	
Event	**ALTER QMGR parameter**
Authority Inhibit Local Remote Start and Stop	AUTHOREV (ENABLED) INHIBTEV (ENABLED) LOCALEV (ENABLED) REMOTEEV (ENABLED) STRSTPEV (ENABLED)

Controlling channel and bridge events

You control channel events by using queue manager attributes. To enable channel events, set the appropriate queue manager attribute to ENABLED. To disable channel events, set the appropriate queue manager attribute to DISABLED.

To enable or disable channels events use the MQSC command ALTER QMGR, specifying the appropriate queue manager attribute. summarizes how you enable channel and bridge events. To disable a queue manager event, set the appropriate parameter to DISABLED.

Table 2. Enabling channel and bridge events using MQSC commands	
Event	**ALTER QMGR parameter**
Channel Related to channel errors only IMS Bridge SSL Channel auto-definition	CHLEV (ENABLED) CHLEV (EXCEPTION) BRIDGEEV (ENABLED) SSLEV (ENABLED) CHADEV(ENABLED)

With CHLEV set to exception, the following return codes, and corresponding reason qualifiers are generated:

- MQRC_CHANNEL_ACTIVATED
- MQRC_CHANNEL_CONV_ERROR
- MQRC_CHANNEL_NOT_ACTIVATED
- MQRC_CHANNEL_STOPPED
  - with the following ReasonQualifiers:
    - MQRQ_CHANNEL_STOPPED_ERROR
    - MQRQ_CHANNEL_STOPPED_RETRY
    - MQRQ_CHANNEL_STOPPED_DISABLED
- MQRC_CHANNEL_STOPPED_BY_USER
- MQRC_CHANNEL_BLOCKED
  - with the following ReasonQualifiers:
    - MQRQ_CHANNEL_BLOCKED_NOACCESS
    - MQRQ_CHANNEL_BLOCKED_USERID

- MQRQ_CHANNEL_BLOCKED_ADDRESS

Controlling performance events

You control performance events using the PERFMEV queue manager attribute. To enable performance events, set PERFMEV to ENABLED. To disable performance events, set the PERFMEV queue manager attribute to DISABLED.

To set the PERFMEV queue manager attribute to ENABLED, use the following MQSC command:

```
ALTER QMGR PERFMEV (ENABLED)
```

To enable specific performance events, set the appropriate queue attribute. Also, specify the conditions that cause the event.

Queue depth events

By default, all queue depth events are disabled. To configure a queue for any of the queue depth events:

1. Enable performance events on the queue manager.

2. Enable the event on the required queue.

3. Set the limits, if required, to the appropriate levels, expressed as a percentage of the maximum queue depth.

Queue service interval events

To configure a queue for queue service interval events you must:

1. Enable performance events on the queue manager.

2. Set the control attribute for a Queue Service Interval High or OK event on the queue as required.

3. Specify the service interval time by setting the QSVCINT attribute for the queue to the appropriate length of time.

Note: When enabled, a queue service interval event can be generated at any appropriate time, not necessarily waiting until an MQI call for the queue is issued. However, if an MQI call is used on a queue to put or remove a message, any applicable performance event is generated at that time. The event is *not* generated when the elapsed time becomes equal to the service interval time.

Controlling configuration, command, and logger events

You control configuration, command, and logger events by using the queue manager attributes CONFIGEV, CMDEV, and LOGGEREV. To enable these events, set the appropriate queue manager attribute to ENABLED. To disable these events, set the appropriate queue manager attribute to DISABLED.

Configuration events

To enable configuration events, set CONFIGEV to ENABLED. To disable configuration events, set CONFIGEV to DISABLED. For example, you can enable configuration events by using the following MQSC command:

```
ALTER QMGR CONFIGEV (ENABLED)
```

Command events

To enable command events, set CMDEV to ENABLED. To enable command events for commands except DISPLAY MQSC commands and Inquire PCF commands, set the CMDEV to NODISPLAY. To disable command events, set CMDEV to DISABLED. For example, you can enable command events by using the following MQSC command:

```
ALTER QMGR CMDEV (ENABLED)
```

Logger events

To enable logger events, set LOGGEREV to ENABLED. To disable logger events, set LOGGEREV to DISABLED. For example, you can enable logger events by using the following MQSC command:

```
ALTER QMGR LOGGEREV(ENABLED)
```

Event queues

When an event occurs, the queue manager puts an event message on the defined event queue. The event message contains information about the event.

You can define event queues either as local queues, alias queues, or as local definitions of remote queues. If you define all your event queues as local definitions of the same remote queue on one queue manager, you can centralize your monitoring activities.

You must not define event queues as transmission queues, because event messages have formats that are incompatible with the message format that is required for transmission queues.

Shared event queues are local queues defined with the QSGDISP(SHARED) value.

When an event queue is unavailable

If an event occurs when the event queue is not available, the event message is lost. For example, if you do not define an event queue for a category of event, all event messages for that category are lost. The event messages are not, for example, saved on the dead-letter (undelivered-message) queue.

However, you can define the event queue as a remote queue. Then, if there is a problem on the remote system putting messages to the resolved queue, the event message arrives on the dead-letter queue of the remote system.

An event queue might be unavailable for many different reasons including:

- The queue has not been defined.
- The queue has been deleted.
- The queue is full.
- The queue has been put-inhibited.

The absence of an event queue does not prevent the event from occurring. For example, after a performance event, the queue manager changes the queue attributes and resets the queue statistics. This change happens whether the event message is put on the performance event queue or not. The same is true in the case of configuration and command events.

Using triggered event queues

You can set up the event queues with triggers so that when an event is generated, the event message being put onto the event queue starts a user-written monitoring application. This application can process the event messages and take appropriate action. For example, certain events might require an operator to be informed, other events might start an application that performs some administration tasks automatically.

Event queues can have trigger actions associated with them and can create trigger messages. However, if these trigger messages in turn cause conditions that would normally generate an event, no event is generated. not generating an event in this instance ensures that looping does not occur.

Related concepts

"Controlling events" on page 15
You enable and disable events by specifying the appropriate values for queue manager, queue attributes, or both, depending on the type of event.

"Format of event messages" on page 19

Event messages contain information about an event and its cause. Like other WebSphere MQ messages, an event message has two parts: a message descriptor and the message data.

Conditions for a trigger event
Related reference
QSGDisp (MQLONG)

Format of event messages

Event messages contain information about an event and its cause. Like other WebSphere MQ messages, an event message has two parts: a message descriptor and the message data.

- The message descriptor is based on the MQMD structure.

- The message data consists of an *event header* and the *event data*. The event header contains the reason code that identifies the event type. Putting the event message, and any subsequent action, does not affect the reason code returned by the MQI call that caused the event. The event data provides further information about the event.

Typically, you process event messages with a system management application tailored to meet the requirements of the enterprise at which it runs.

When the queue managers in a queue sharing group detect the conditions for generating an event message, several queue managers can generate an event message for the shared queue, resulting in several event messages. To ensure that a system can correlate multiple event messages from different queue managers, these event messages have a unique correlation identifier *(CorrelId)* set in the message descriptor (MQMD).

Related reference
"Activity report MQMD (message descriptor)" on page 96
Use this page to view the values contained by the MQMD structure for an activity report

"Activity report MQEPH (Embedded PCF header)" on page 100
Use this page to view the values contained by the MQEPH structure for an activity report

"Activity report MQCFH (PCF header)" on page 101
Use this page to view the PCF values contained by the MQCFH structure for an activity report

Event message reference
Event message format
Event message MQMD (message descriptor)
Event message MQCFH (PCF header)
Event message descriptions

Performance events

Performance events relate to conditions that can affect the performance of applications that use a specified queue. The scope of performance events is the queue. **MQPUT** calls and **MQGET** calls on one queue do not affect the generation of performance events on another queue.

Performance event messages can be generated at any appropriate time, not necessarily waiting until an MQI call for the queue is issued. However, if you use an MQI call on a queue to put or remove a message, any appropriate performance events are generated at that time.

Every performance event message that is generated is placed on the queue, SYSTEM.ADMIN.PERFM.EVENT.

The event data contains a reason code that identifies the cause of the event, a set of performance event statistics, and other data. The types of event data that can be returned in performance event messages are described in the following list:

- Queue Depth High
- Queue Depth Low

- Queue Full
- Queue Service Interval High
- Queue Service Interval OK

Examples that illustrate the use of performance events assume that you set queue attributes by using the appropriate IBM WebSphere MQ commands (MQSC). On , you can also set queue attributes using the operations and controls panels for queue managers.

Related reference
"Event types" on page 8
Use this page to view the types of instrumentation event that a queue manager or channel instance can report

Performance event statistics

The performance event data in the event message contains statistics about the event. Use the statistics to analyze the behavior of a specified queue.

The event data in the event message contains information about the event for system management programs. For all performance events, the event data contains the names of the queue manager and the queue associated with the event. The event data also contains statistics related to the event. Table 3 on page 20 summarizes the event statistics that you can use to analyze the behavior of a queue. All the statistics refer to what has happened since the last time the statistics were reset.

Table 3. Performance event statistics	
Parameter	**Description**
TimeSinceReset	The elapsed time since the statistics were last reset.
HighQDepth	The maximum number of messages on the queue since the statistics were last reset.
MsgEnqCount	The number of messages enqueued (the number of MQPUT calls to the queue), since the statistics were last reset.
MsgDeqCount	The number of messages dequeued (the number of MQGET calls to the queue), since the statistics were last reset.

Performance event statistics are reset when any of the following changes occur:

- A performance event occurs (statistics are reset on all active queue managers).
- A queue manager stops and restarts.
- The PCF command, Reset Queue Statistics, is issued from an application program.

Related concepts
"Performance events" on page 19
Performance events relate to conditions that can affect the performance of applications that use a specified queue. The scope of performance events is the queue. **MQPUT** calls and **MQGET** calls on one queue do not affect the generation of performance events on another queue.

"The service timer" on page 22
Queue service interval events use an internal timer, called the *service timer*, which is controlled by the queue manager. The service timer is used only if a queue service interval event is enabled.

"Rules for queue service interval events" on page 23
Formal rules control when the service timer is set and queue service interval events are generated.

Related tasks
"Enabling queue service interval events" on page 23

To configure a queue for queue service interval events you set the appropriate queue manager and queue attributes.

Related reference

Queue Depth High

Reset Queue Statistics

Queue service interval events

Queue service interval events indicate whether an operation was performed on a queue within a user-defined time interval called the *service interval*. Depending on your installation, you can use queue service interval events to monitor whether messages are being taken off queues quickly enough.

Queue service interval events are *not* supported on shared queues.

The following types of queue service interval events can occur, where the term *get operation* refers to an **MQGET** call or an activity that removes a messages from a queue, such as using the **CLEAR QLOCAL** command:

Queue Service Interval OK
> Indicates that after one of the following operations:
>
> - An MQPUT call
> - A get operation that leaves a non-empty queue
>
> a get operation was performed within a user-defined time period, known as the *service interval*.
>
> Only a get operation can cause the Queue Service Interval OK event message. Queue Service Interval OK events are sometimes described as OK events.

Queue Service Interval High
> Indicates that after one of the following operations:
>
> - An MQPUT call
> - A get operation that leaves a non-empty queue
>
> a get operation was **not** performed within a user-defined service interval.
>
> Either a get operation or an MQPUT call can cause the Queue Service Interval High event message. Queue Service Interval High events are sometimes described as High events.

To enable both Queue Service Interval OK and Queue Service Interval High events, set the QServiceIntervalEvent control attribute to High. Queue Service Interval OK events are automatically enabled when a Queue Service Interval High event is generated. You do not need to enable Queue Service Interval OK events independently.

OK and High events are mutually exclusive, so if one is enabled the other is disabled. However, both events can be simultaneously disabled.

Figure 3 on page 22 shows a graph of queue depth against time. At time P1, an application issues an MQPUT, to put a message on the queue. At time G1, another application issues an MQGET to remove the message from the queue.
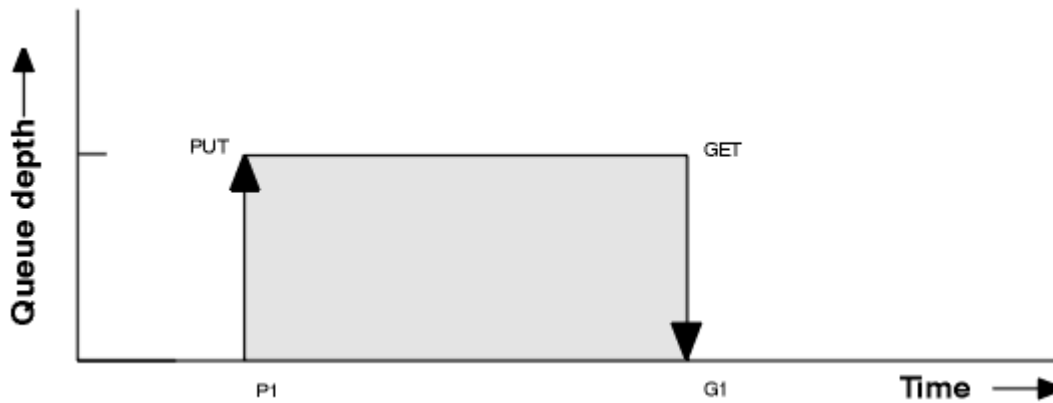
Figure 3. Understanding queue service interval events

The possible outcomes of queue service interval events are as follows:

- If the elapsed time between the put and the get is less than or equal to the service interval:
  - A *Queue Service Interval OK* event is generated at time G1, if queue service interval events are enabled
- If the elapsed time between the put and get is greater than the service interval:
  - A *Queue Service Interval High* event is generated at time G1, if queue service interval events are enabled.

The algorithm for starting the service timer and generating events is described in "Rules for queue service interval events" on page 23.

Related reference

Queue Service Interval OK
Queue Service Interval High
QServiceIntervalEvent (MQLONG)
ServiceIntervalEvent property

The service timer

Queue service interval events use an internal timer, called the *service timer*, which is controlled by the queue manager. The service timer is used only if a queue service interval event is enabled.

What precisely does the service timer measure?
The service timer measures the elapsed time between an MQPUT call to an empty queue or a get operation, and the next put or get, provided the queue depth is nonzero between these two operations.

When is the service timer active?
The service timer is always active (running), if the queue has messages on it (depth is nonzero) and a queue service interval event is enabled. If the queue becomes empty (queue depth zero), the timer is put into an OFF state, to be restarted on the next put.

When is the service timer reset?
The service timer is always reset after a get operation . It is also reset by an MQPUT call to an empty queue. However, it is not necessarily reset on a queue service interval event.

How is the service timer used?
Following a get operation or an MQPUT call, the queue manager compares the elapsed time as measured by the service timer, with the user-defined service interval. The result of this comparison is that:

- An OK event is generated if there is a get operation and the elapsed time is less than or equal to the service interval, AND this event is enabled.

- A high event is generated if the elapsed time is greater than the service interval, AND this event is enabled.

Can applications read the service timer?
No, the service timer is an internal timer that is not available to applications.

What about the *TimeSinceReset* parameter?
The *TimeSinceReset* parameter is returned as part of the event statistics in the event data. It specifies the time between successive queue service interval events, unless the event statistics are reset.

Rules for queue service interval events

Formal rules control when the service timer is set and queue service interval events are generated.

Rules for the service timer

The service timer is reset to zero and restarted as follows:

- After an MQPUT call to an empty queue.
- After an MQGET call, if the queue is not empty after the MQGET call.

The resetting of the timer does not depend on whether an event has been generated.

At queue manager startup the service timer is set to startup time if the queue depth is greater than zero.

If the queue is empty following a get operation, the timer is put into an OFF state.

Queue Service Interval High events

The Queue Service Interval event must be enabled (set to HIGH).

Queue Service Interval High events are automatically enabled when a Queue Service Interval OK event is generated.

If the service time is greater than the service interval, an event is generated on, or before, the next MQPUT or get operation.

Queue Service Interval OK events

Queue Service Interval OK events are automatically enabled when a Queue Service Interval High event is generated.

If the service time (elapsed time) is less than or equal to the service interval, an event is generated on, or before, the next get operation.

Related tasks
"Enabling queue service interval events" on page 23
To configure a queue for queue service interval events you set the appropriate queue manager and queue attributes.

Enabling queue service interval events

To configure a queue for queue service interval events you set the appropriate queue manager and queue attributes.

About this task

The high and OK events are mutually exclusive; that is, when one is enabled, the other is automatically disabled:

- When a high event is generated on a queue, the queue manager automatically disables high events and enables OK events for that queue.

- When an OK event is generated on a queue, the queue manager automatically disables OK events and enables high events for that queue.

Table 4. Enabling queue service interval events using MQSC	
Queue service interval event	**Queue attributes**
Queue Service Interval High Queue Service Interval OK No queue service interval events	QSVCIEV (HIGH) QSVCIEV (OK) QSVCIEV (NONE)
Service interval	QSVCINT (tt) where tt is the service interval time in milliseconds.

Perform the following steps to enable queue service interval events:

Procedure

1. Set the queue manager attribute PERFMEV to ENABLED.

   Performance events are enabled on the queue manager.
2. Set the control attribute, QSVCIEV, for a Queue Service Interval High or OK event on the queue, as required.
3. Set the QSVCINT attribute for the queue to specify the appropriate service interval time.

Example

To enable Queue Service Interval High events with a service interval time of 10 seconds (10 000 milliseconds) use the following MQSC commands:

```
ALTER QMGR PERFMEV(ENABLED)

ALTER QLOCAL('MYQUEUE') QSVCINT(10000) QSVCIEV(HIGH)
```

.

Queue service interval events examples

Use these examples to understand the information that you can obtain from queue service interval events

The three examples provide progressively more complex illustrations of the use of queue service interval events.

The figures accompanying the examples have the same structure:

- Figure 1 is a graph of queue depth against time, showing individual MQGET calls and MQPUT calls.
- The Commentary section shows a comparison of the time constraints. There are three time periods that you must consider:
  - The user-defined service interval.
  - The time measured by the service timer.
  - The time since event statistics were last reset (TimeSinceReset in the event data).
- The Event statistics summary section shows which events are enabled at any instant and what events are generated.

The examples illustrate the following aspects of queue service interval events:

- How the queue depth varies over time.
- How the elapsed time as measured by the service timer compares with the service interval.

- Which event is enabled.
- Which events are generated.

Remember: Example 1 shows a simple case where the messages are intermittent and each message is removed from the queue before the next one arrives. From the event data, you know that the maximum number of messages on the queue was one. You can, therefore, work out how long each message was on the queue.

However, in the general case, where there is more than one message on the queue and the sequence of MQGET calls and MQPUT calls is not predictable, you cannot use queue service interval events to calculate how long an individual message remains on a queue. The TimeSinceReset parameter, which is returned in the event data, can include a proportion of time when there are no messages on the queue. Therefore any results you derive from these statistics are implicitly averaged to include these times.

Related concepts
"Queue service interval events" on page 21
Queue service interval events indicate whether an operation was performed on a queue within a user-defined time interval called the *service interval*. Depending on your installation, you can use queue service interval events to monitor whether messages are being taken off queues quickly enough.

"The service timer" on page 22

Queue service interval events use an internal timer, called the *service timer*, which is controlled by the queue manager. The service timer is used only if a queue service interval event is enabled.

Queue service interval events: example 1

A basic sequence of MQGET calls and MQPUT calls, where the queue depth is always one or zero.
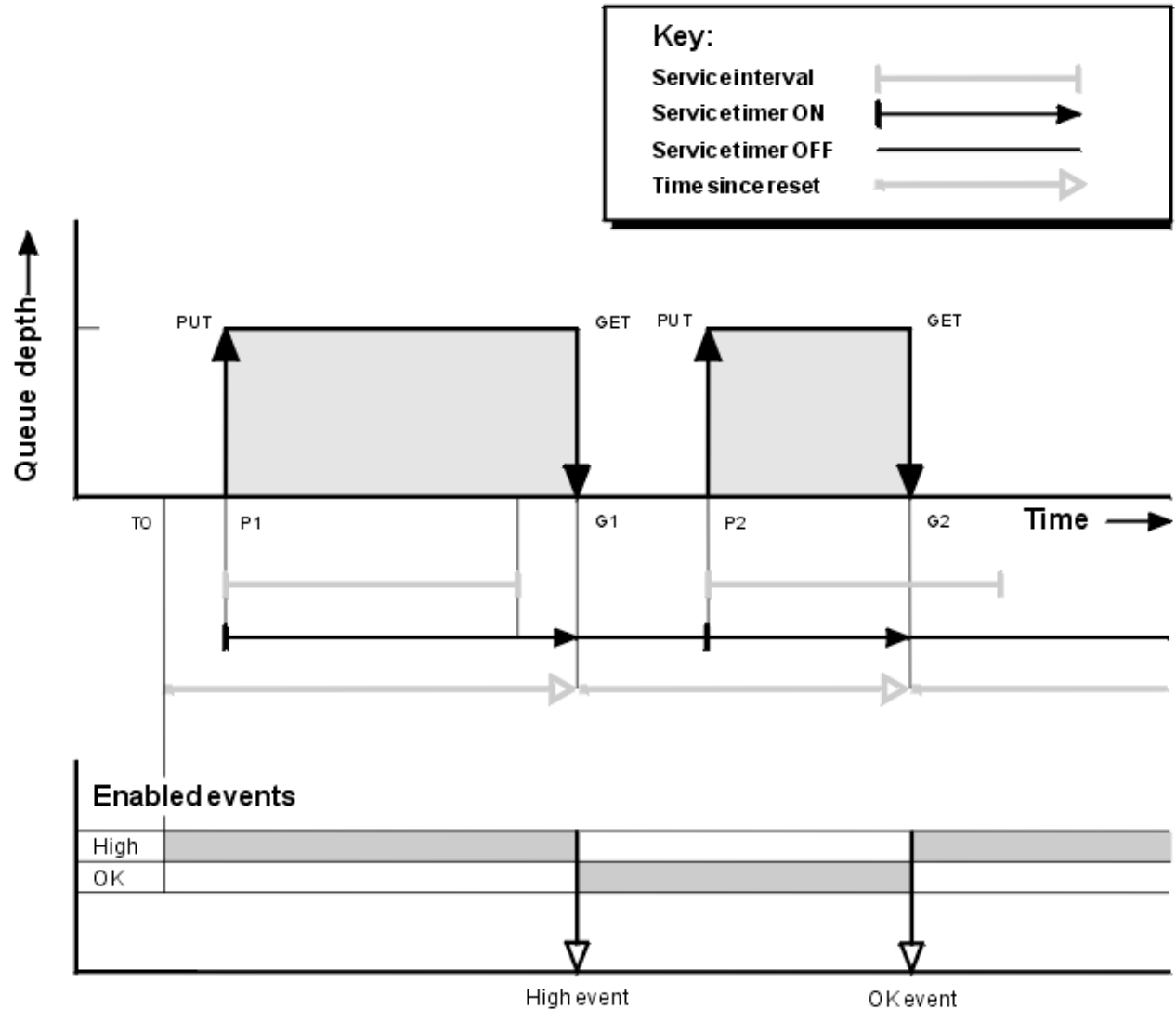
Figure 4. Queue service interval events - example 1

Commentary

1. At P1, an application puts a message onto an empty queue. This starts the service timer.

   Note that T0 might be queue manager startup time.

2. At G1, another application gets the message from the queue. Because the elapsed time between P1 and G1 is greater than the service interval, a Queue Service Interval High event is generated on the MQGET call at G1. When the high event is generated, the queue manager resets the event control attribute so that:

   a. The OK event is automatically enabled.

   b. The high event is disabled.

   Because the queue is now empty, the service timer is switched to an OFF state.

3. At P2, a second message is put onto the queue. This restarts the service timer.

4. At G2, the message is removed from the queue. However, because the elapsed time between P2 and G2 is less than the service interval, a Queue Service Interval OK event is generated on the MQGET call at G2. When the OK event is generated, the queue manager resets the control attribute so that:

   a. The high event is automatically enabled.
   b. The OK event is disabled.

   Because the queue is empty, the service timer is again switched to an OFF state.

Event statistics summary

Table 5 on page 27 summarizes the event statistics for this example.

Table 5. Event statistics summary for example 1		
	Event 1	**Event 2**
Time of event	T(G1)	T(G2)
Type of event	High	OK
TimeSinceReset	T(G1) - T(0)	T(G2) - T(G1)
HighQDepth	1	1
MsgEnqCount	1	1
MsgDeqCount	1	1

The middle part of Figure 4 on page 26 shows the elapsed time as measured by the service timer compared to the service interval for that queue. To see whether a queue service interval event might occur, compare the length of the horizontal line representing the service timer (with arrow) to that of the line representing the service interval. If the service timer line is longer, and the Queue Service Interval High event is enabled, a Queue Service Interval High event occurs on the next get. If the timer line is shorter, and the Queue Service Interval OK event is enabled, a Queue Service Interval OK event occurs on the next get.

Queue service interval events: example 2
A sequence of MQPUT calls and MQGET calls, where the queue depth is not always one or zero.

This example also shows instances of the timer being reset without events being generated, for example, at time P2.
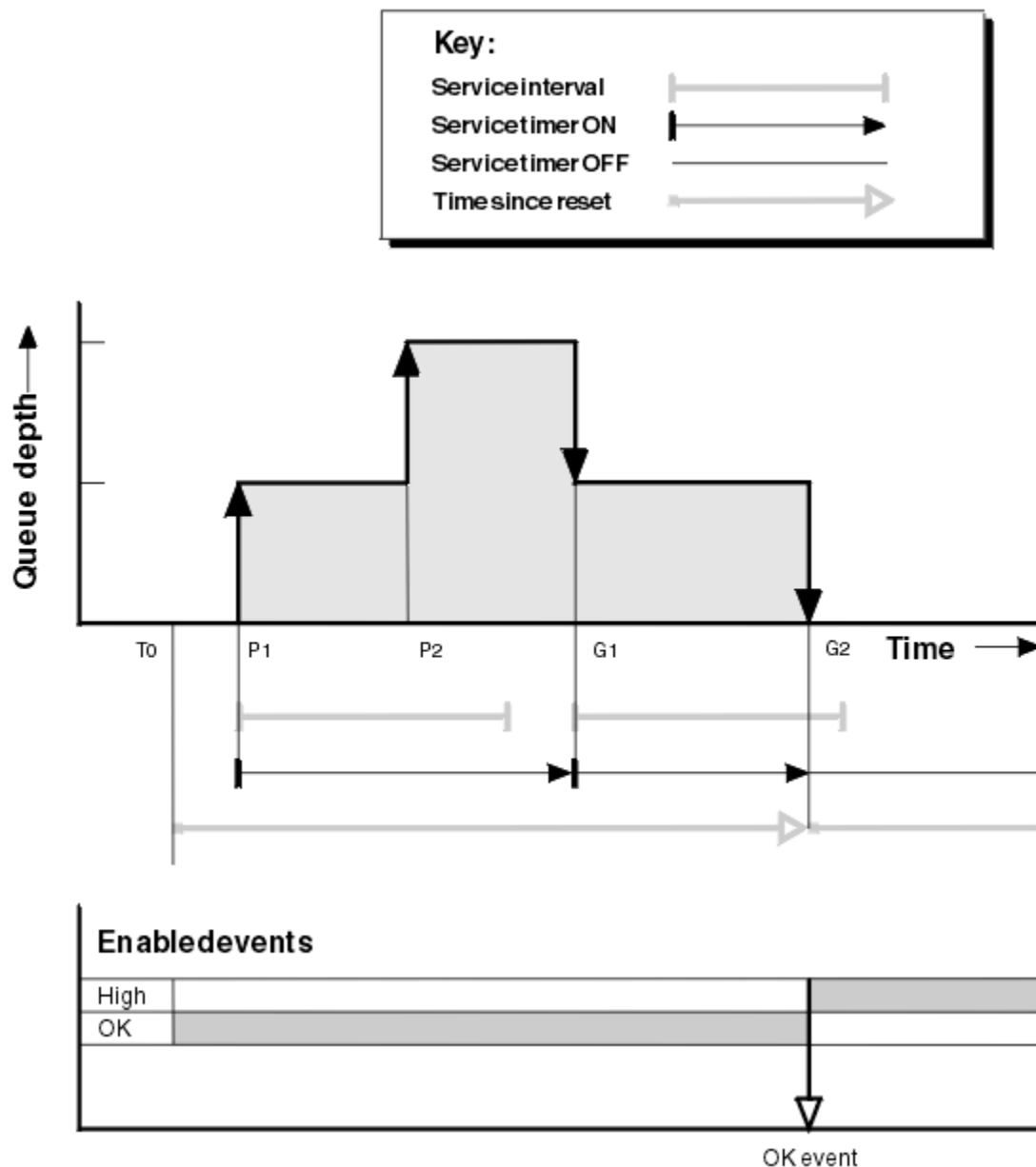
Figure 5. Queue service interval events - example 2

Commentary

In this example, OK events are enabled initially and queue statistics were reset at time T0.

1. At P1, the first put starts the service timer.

2. At P2, the second put does not generate an event because a put cannot cause an OK event.

3. At G1, the service interval has now been exceeded and therefore an OK event is not generated. However, the MQGET call causes the service timer to be reset.

4. At G2, the second get occurs within the service interval and this time an OK event is generated. The queue manager resets the event control attribute so that:

   a. The high event is automatically enabled.

   b. The OK event is disabled.

   Because the queue is now empty, the service timer is switched to an OFF state.

Event statistics summary

Table 6 on page 29 summarizes the event statistics for this example.

Table 6. Event statistics summary for example 2	
	Event 2
Time of event	T(G2)
Type of event	OK
TimeSinceReset	T(G2) - T(0)
HighQDepth	2
MsgEnqCount	2
MsgDeqCount	2

Queue service interval events: example 3
A sequence of MQGET calls and MQPUT calls that is more sporadic than the previous examples.
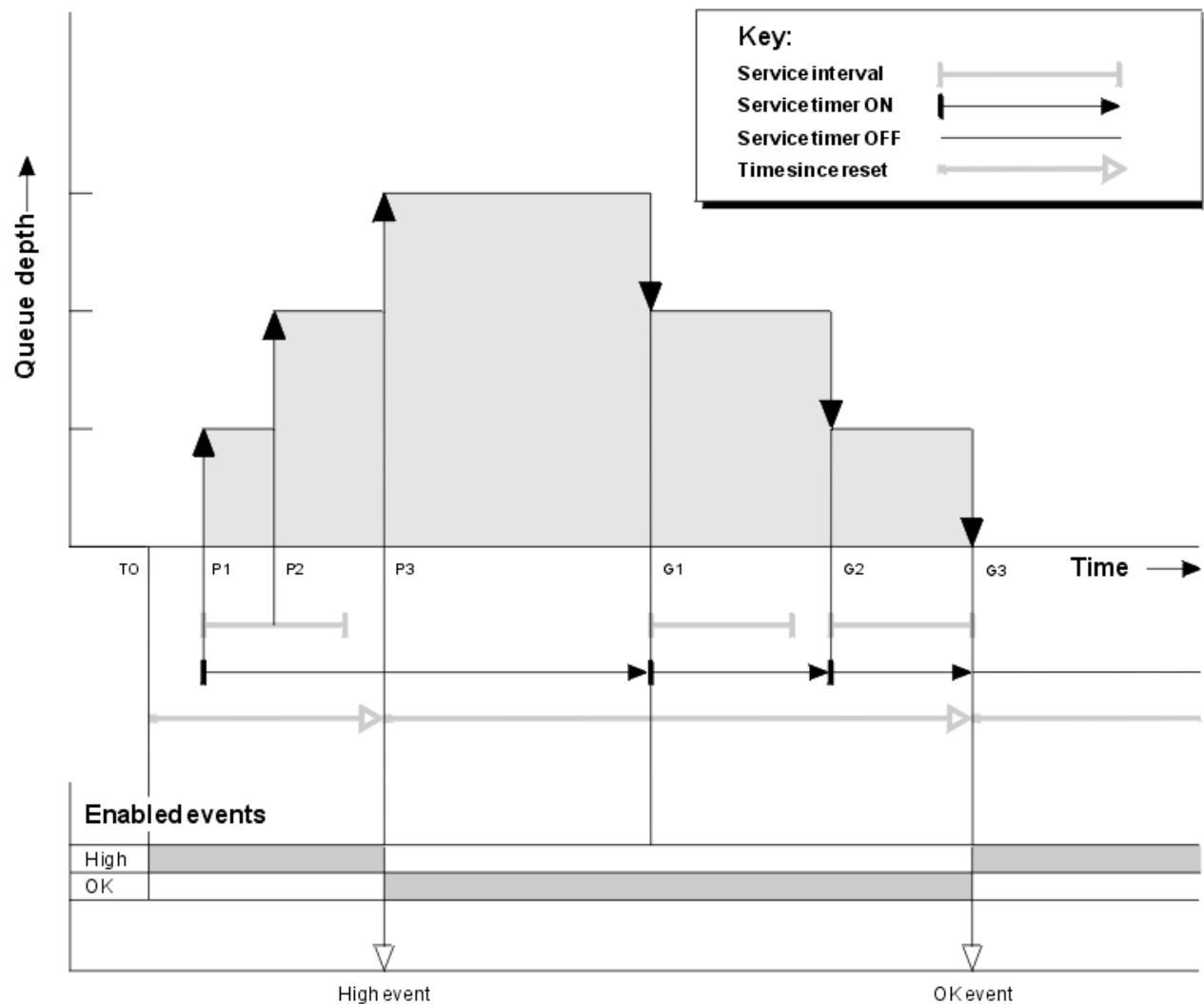
Figure 6. Queue service interval events - example 3

Commentary

1. At time T(0), the queue statistics are reset and Queue Service Interval High events are enabled.
2. At P1, the first put starts the service timer.
3. At P2, the second put increases the queue depth to two. A high event is not generated here because the service interval time has not been exceeded.
4. At P3, the third put causes a high event to be generated. (The timer has exceeded the service interval.) The timer is not reset because the queue depth was not zero before the put. However, OK events are enabled.
5. At G1, the MQGET call does not generate an event because the service interval has been exceeded and OK events are enabled. The MQGET call does, however, reset the service timer.
6. At G2, the MQGET call does not generate an event because the service interval has been exceeded and OK events are enabled. Again, the MQGET call resets the service timer.
7. At G3, the third get empties the queue and the service timer is *equal* to the service interval. Therefore an OK event is generated. The service timer is reset and high events are enabled. The MQGET call empties the queue, and this puts the timer in the OFF state.

Event statistics summary

Table 7 on page 30 summarizes the event statistics for this example.

Table 7. Event statistics summary for example 3		
	Event 1	**Event 2**
Time of event	T(P3)	T(G3)
Type of event	High	OK
TimeSinceReset	T(P3) - T(0)	T(G3) - T(P3)
HighQDepth	3	3
MsgEnqCount	3	0
MsgDeqCount	0	3

Queue depth events

Queue depth events are related to the queue depth, that is, the number of messages on the queue.

In WebSphere MQ applications, queues must not become full. If they do, applications can no longer put messages on the queue that they specify. Although the message is not lost if this occurs, a full queue can cause considerable inconvenience. The number of messages can build up on a queue if the messages are being put onto the queue faster than the applications that process them can take them off.

The solution to this problem depends on the particular circumstances, but might involve:

- Diverting some messages to another queue.
- Starting new applications to take more messages off the queue.
- Stopping nonessential message traffic.
- Increasing the queue depth to overcome a transient maximum.

Advance warning that problems might be on their way makes it easier to take preventive action. For this purpose, WebSphere MQ provides the following queue depth events:

Queue Depth High events
Indicate that the queue depth has increased to a predefined threshold called the Queue Depth High limit.

Queue Depth Low events
> Indicate that the queue depth has decreased to a predefined threshold called the Queue Depth Low limit.

Queue Full events
> Indicate that the queue has reached its maximum depth, that is, the queue is full.

A Queue Full Event is generated when an application attempts to put a message on a queue that has reached its maximum depth. Queue Depth High events give advance warning that a queue is filling up. This means that having received this event, the system administrator needs to take some preventive action. You can configure the queue manager such that, if the preventive action is successful and the queue depth drops to a safer level, the queue manager generates a Queue Depth Low event.

The first queue depth event example illustrates the effect of presumed action preventing the queue becoming full.

Related concepts
"Queue depth events examples" on page 32
Use these examples to understand the information that you can obtain from queue depth events

Related reference
Queue Full
Queue Depth High
Queue Depth Low

Enabling queue depth events

To configure a queue for any of the queue depth events you set the appropriate queue manager and queue attributes.

About this task

By default, all queue depth events are disabled. When enabled, queue depth events are generated as follows:

- A Queue Depth High event is generated when a message is put on the queue, causing the queue depth to be greater than or equal to the value determined by the Queue Depth High limit.

  - A Queue Depth High event is automatically enabled by a Queue Depth Low event on the same queue.
  - A Queue Depth High event automatically enables both a Queue Depth Low and a Queue Full event on the same queue.

- A Queue Depth Low event is generated when a message is removed from a queue by a get operation causing the queue depth to be less than or equal to the value determined by the Queue Depth Low limit.

  - A Queue Depth Low event is automatically enabled by a Queue Depth High event or a Queue Full event on the same queue.
  - A Queue Depth Low event automatically enables both a Queue Depth High and a Queue Full event on the same queue.

- A Queue Full event is generated when an application is unable to put a message onto a queue because the queue is full.

  - A Queue Full event is automatically enabled by a Queue Depth High or a Queue Depth Low event on the same queue.
  - A Queue Full event automatically enables a Queue Depth Low event on the same queue.

Perform the following steps to configure a queue for any of the queue depth events:

Procedure

1. Enable performance events on the queue manager, using the queue manager attribute PERFMEV.
2. Set one of the following attributes to enable the event on the required queue:

- *QDepthHighEvent* (QDPHIEV in MQSC)
- *QDepthLowEvent* (QDPLOEV in MQSC)
- *QDepthMaxEvent* (QDPMAXEV in MQSC)

3. Optional: To set the limits, assign the following attributes, as a percentage of the maximum queue depth:

- *QDepthHighLimit* (QDEPTHHI in MQSC)
- *QDepthLowLimit* (QDEPTHLO in MQSC)

Restriction: QDEPTHHI must not be less than QDEPTHLO.

If QDEPTHHI equals QDEPTHLO an event message is generated every time the queue depth passes the value in either direction, because the high threshold is enabled when the queue depth is below the value and the low threshold is enabled when the depth is above the value.

Results

Note:

A Queue Depth Low event is not generated when expired messages are removed from a queue by a get operation causing the queue depth to be less than, or equal to, the value determined by the Queue Depth Low limit.

IBM WebSphere MQgenerates the low event message only during a successful get operation. Therefore, when the expired messages are removed from the queue, no queue depth low event message is generated.

Additionally, after the removal of these expired messages from the queue, queue depth high event and queue depth low event are not reset.

Example

To enable Queue Depth High events on the queue MYQUEUE with a limit set at 80%, use the following MQSC commands:

```
ALTER QMGR PERFMEV(ENABLED)
ALTER QLOCAL('MYQUEUE') QDEPTHHI(80) QDPHIEV(ENABLED)
```

To enable Queue Depth Low events on the queue MYQUEUE with a limit set at 20%, use the following MQSC commands:

```
ALTER QMGR PERFMEV(ENABLED)
ALTER QLOCAL('MYQUEUE') QDEPTHLO(20) QDPLOEV(ENABLED)
```

To enable Queue Full events on the queue MYQUEUE, use the following MQSC commands:

```
ALTER QMGR PERFMEV(ENABLED)
ALTER QLOCAL('MYQUEUE') QDPMAXEV(ENABLED)
```

Queue depth events examples

Use these examples to understand the information that you can obtain from queue depth events

The first example provides a basic illustration of queue depth events. The second example is more extensive, but the principles are the same as for the first example. Both examples use the same queue definition, as follows:

The queue, MYQUEUE1, has a maximum depth of 1000 messages. The high queue depth limit is 80% and the low queue depth limit is 20%. Initially, Queue Depth High events are enabled, while the other queue depth events are disabled.

The WebSphere MQ commands (MQSC) to configure this queue are:

```
ALTER QMGR PERFMEV(ENABLED)

DEFINE QLOCAL('MYQUEUE1') MAXDEPTH(1000) QDPMAXEV(DISABLED) QDEPTHHI(80)
  QDPHIEV(ENABLED) QDEPTHLO(20) QDPLOEV(DISABLED)
```

Related concepts

"Queue depth events" on page 30
Queue depth events are related to the queue depth, that is, the number of messages on the queue.

Related tasks

"Enabling queue depth events" on page 31
To configure a queue for any of the queue depth events you set the appropriate queue manager and queue attributes.

Related reference

The MQSC commands

Queue depth events: example 1

A basic sequence of queue depth events.

Figure 7 on page 33 shows the variation of queue depth over time.
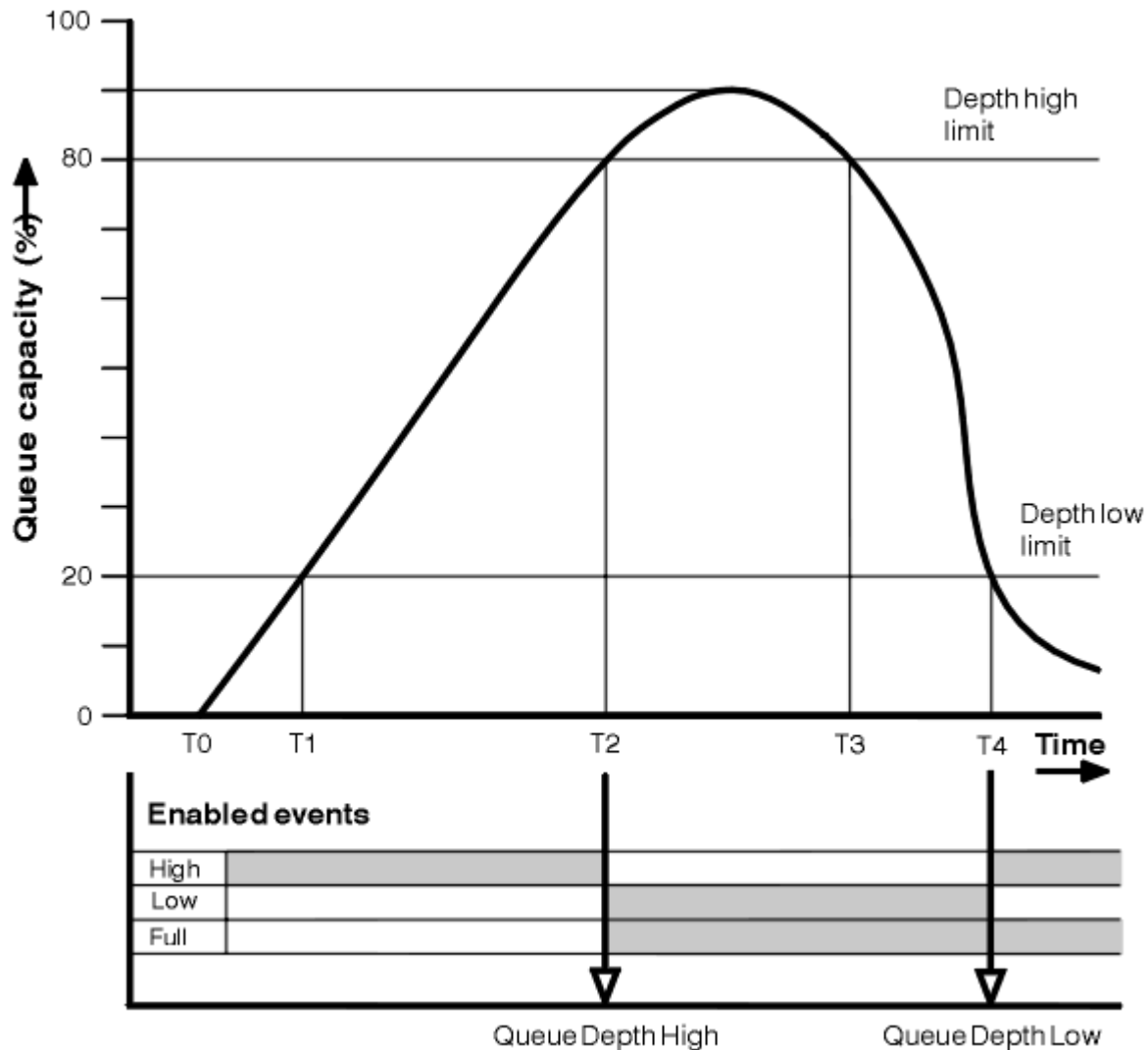
Figure 7. Queue depth events (1)

Commentary

1. At T(1), the queue depth is increasing (more MQPUT calls than MQGET calls) and crosses the Queue Depth Low limit. No event is generated at this time.

2. The queue depth continues to increase until T(2), when the depth high limit (80%) is reached and a Queue Depth High event is generated.

   This enables both Queue Full and Queue Depth Low events.

3. The (presumed) preventive actions instigated by the event prevent the queue from becoming full. By time T(3), the Queue Depth High limit has been reached again, this time from above. No event is generated at this time.

4. The queue depth continues to fall until T(4), when it reaches the depth low limit (20%) and a Queue Depth Low event is generated.

   This enables both Queue Full and Queue Depth High events.

Event statistics summary

Table 8 on page 34 summarizes the queue event statistics and Table 9 on page 34 summarizes which events are enabled.

Table 8. Event statistics summary for queue depth events (example 1)

	Event 2	Event 4
Time of event	T(2)	T(4)
Type of event	Queue Depth High	Queue Depth Low
TimeSinceReset	T(2) - T(0)	T(4) - T(2)
HighQDepth (Maximum queue depth since reset)	800	900
MsgEnqCount	1157	1220
MsgDeqCount	357	1820

Table 9. Summary showing which events are enabled

Time period	Queue Depth High event	Queue Depth Low event	Queue Full event
Before T(1)	ENABLED	-	-
T(1) to T(2)	ENABLED	-	-
T(2) to T(3)	-	ENABLED	ENABLED
T(3) to T(4)	-	ENABLED	ENABLED
After T(4)	ENABLED	-	ENABLED

Queue depth events: example 2
A more extensive sequence of queue depth events.

Figure 8 on page 35 shows the variation of queue depth over time.
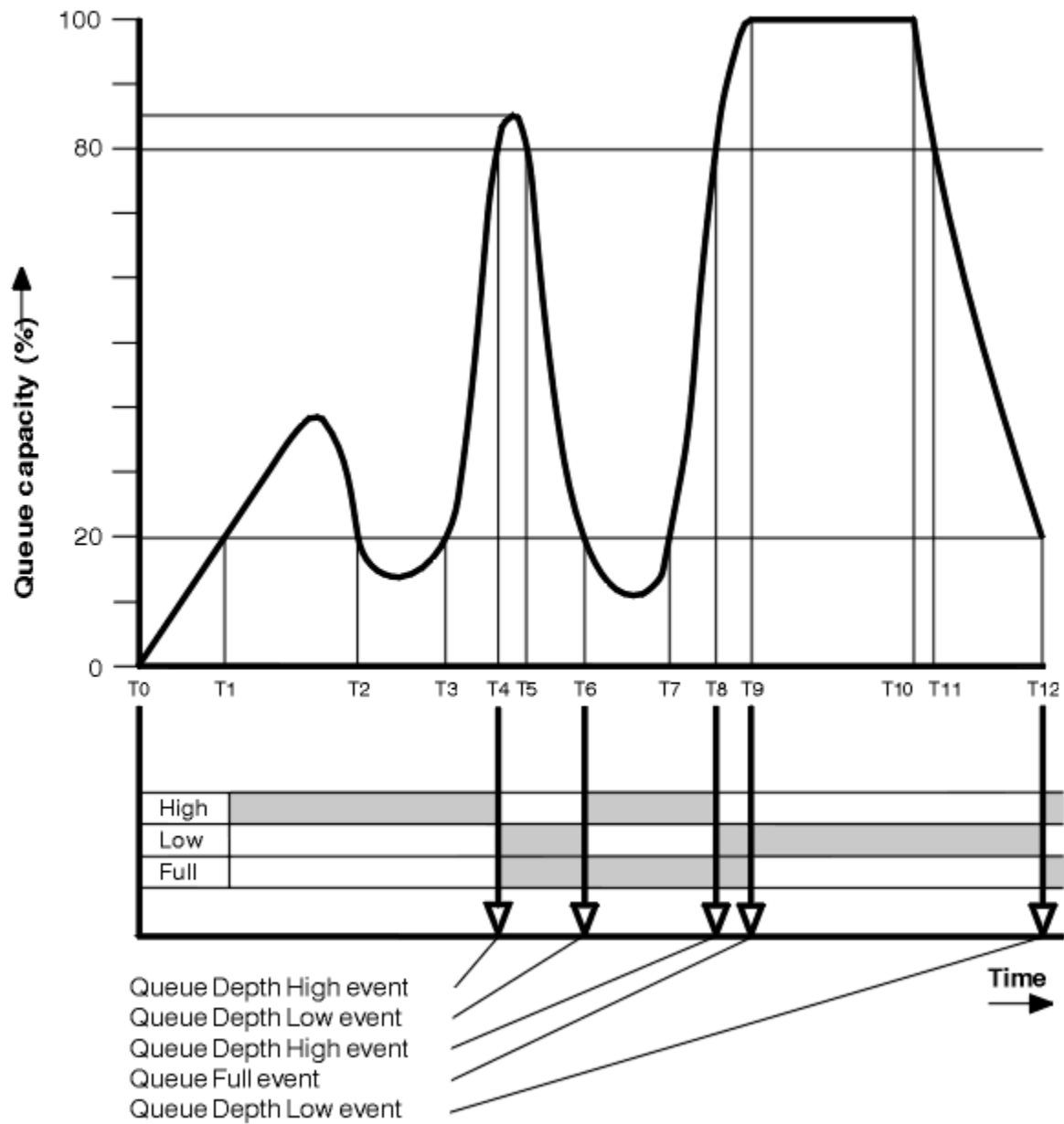
Figure 8. Queue depth events (2)

Commentary

1. No Queue Depth Low event is generated at the following times:
   - T(1) (Queue depth increasing, and not enabled)
   - T(2) (Not enabled)
   - T(3) (Queue depth increasing, and not enabled)
2. At T(4) a Queue Depth High event occurs. This enables both Queue Full and Queue Depth Low events.
3. At T(9) a Queue Full event occurs **after** the first message that cannot be put on the queue because the queue is full.
4. At T(12) a Queue Depth Low event occurs.

Event statistics summary

Table 10 on page 36 summarizes the queue event statistics and Table 11 on page 36 summarizes which events are enabled at different times for this example.

Table 10. Event statistics summary for queue depth events (example 2)					
	Event 4	**Event 6**	**Event 8**	**Event 9**	**Event 12**
Time of event	T(4)	T(6)	T(8)	T(9)	T(12)
Type of event	Queue Depth High	Queue Depth Low	Queue Depth High	Queue Full	Queue Depth Low
TimeSinceReset	T(4) - T(0)	T(6) - T(4)	T(8) - T(6)	T(9) - T(8)	T(12) - T(9)
HighQDepth	800	855	800	1000	1000
MsgEnqCount	1645	311	1377	324	221
MsgDeqCount	845	911	777	124	1021

Table 11. Summary showing which events are enabled			
Time period	**Queue Depth High event**	**Queue Depth Low event**	**Queue Full event**
T(0) to T(4)	ENABLED	-	-
T(4) to T(6)	-	ENABLED	ENABLED
T(6) to T(8)	ENABLED	-	ENABLED
T(8) to T(9)	-	ENABLED	ENABLED
T(9) to T(12)	-	ENABLED	-
After T(12)	ENABLED	-	ENABLED

Note: Events are out of syncpoint. Therefore you could have an empty queue, then fill it up causing an event, then roll back all of the messages under the control of a syncpoint manager. However, event enabling has been automatically set, so that the next time the queue fills up, no event is generated.

Configuration events

Configuration events are notifications that are generated when an object is created, changed, or deleted, and can also be generated by explicit requests.

Configuration events notify you about changes to the attributes of an object. There are four types of configuration events:

- Create object events
- Change object events
- Delete object events
- Refresh object events

The event data contains the following information:

Origin information
comprises the queue manager from where the change was made, the ID of the user that made the change, and how the change came about, for example by a console command.

Context information
a replica of the context information in the message data from the command message.

Context information is included in the event data only when the command was entered as a message on the SYSTEM.COMMAND.INPUT queue.

Object identity
comprises the name, type and disposition of the object.

Object attributes
comprises the values of all the attributes in the object.

In the case of change object events, two messages are generated, one with the information before the change, the other with the information after.

Every configuration event message that is generated is placed on the queue SYSTEM.ADMIN.CONFIG.EVENT.

Related concepts
"Configuration events" on page 12
Configuration events are generated when a configuration event is requested explicitly, or automatically when an object is created, modified, or deleted.

Related reference
Create object
Change object
Delete object
Refresh object
"Event types" on page 8
Use this page to view the types of instrumentation event that a queue manager or channel instance can report

Configuration event generation

Use this page to view the commands that cause configuration events to be generated and to understand the circumstances in which configuration events are not generated

A configuration event message is put to the configuration event queue when the CONFIGEV queue manager attribute is ENABLED and

- any of the following commands, or their PCF equivalent, are issued:
  - DELETE AUTHINFO
  - DELETE CFSTRUCT
  - DELETE CHANNEL
  - DELETE NAMELIST
  - DELETE PROCESS
  - DELETE QMODEL/QALIAS/QREMOTE
  - DELETE STGCLASS
  - DELETE TOPIC
  - REFRESH QMGR
- any of the following commands, or their PCF equivalent, are issued even if there is no change to the object:
  - DEFINE/ALTER AUTHINFO
  - DEFINE/ALTER CFSTRUCT
  - DEFINE/ALTER CHANNEL
  - DEFINE/ALTER NAMELIST
  - DEFINE/ALTER PROCESS
  - DEFINE/ALTER QMODEL/QALIAS/QREMOTE

- – DEFINE/ALTER STGCLASS
- – DEFINE/ALTER TOPIC
- – DEFINE MAXSMSGS
- – SET CHLAUTH
- – ALTER QMGR, unless the CONFIGEV attribute is DISABLED and is not changed to ENABLED
- any of the following commands, or their PCF equivalent, are issued for a local queue that is not temporary dynamic, even if there is no change to the queue.
  - – DELETE QLOCAL
  - – DEFINE/ALTER QLOCAL
- an MQSET call is issued, other than for a temporary dynamic queue, even if there is no change to the object.

When configuration events are not generated

Configuration events messages are not generated in the following circumstances:

- When a command or an MQSET call fails
- When a queue manager encounters an error trying to put a configuration event on the event queue, in which case the command or MQSET call completes, but no event message is generated
- For a temporary dynamic queue
- When internal changes are made to the TRIGGER queue attribute
- For the configuration event queue SYSTEM.ADMIN.CONFIG.EVENT, except by the REFRESH QMGR command
- For REFRESH/RESET CLUSTER and RESUME/SUSPEND QMGR commands that cause clustering changes
- When Creating or deleting a queue manager

Related concepts

Introduction to Programmable Command Formats

"Configuration events" on page 36

Configuration events are notifications that are generated when an object is created, changed, or deleted, and can also be generated by explicit requests.

Related reference

The MQSC commands

MQSET - Set object attributes

Configuration event usage

Use this page to view how you can use configuration events to obtain information about your system, and to understand the factors, such as CMDSCOPE, that can affect your use of configuration events.

You can use configuration events for the following purposes:

1. To produce and maintain a central configuration repository, from which reports can be produced and information about the structure of the system can be generated.

2. To generate an audit trail. For example, if an object is changed unexpectedly, information regarding who made the alteration and when it was done can be stored.

   This can be particularly useful when command events are also enabled. If an MQSC or PCF command causes a configuration event and a command event to be generated, both event messages will share the same correlation identifier in their message descriptor.

For an MQSET call or any of the following commands:

- DEFINE object
- ALTER object

- DELETE object

if the queue manager attribute CONFIGEV is enabled, but the configuration event message cannot be put on the configuration event queue, for example the event queue has not been defined, the command or MQSET call is executed regardless.

Effects of CMDSCOPE

For commands where CMDSCOPE is used, the configuration event message or messages will be generated on the queue manager or queue managers where the command is executed, not where the command is entered. However, all the origin and context information in the event data will relate to the original command as entered, even where the command using CMDSCOPE is one that has been generated by the source queue manager.

Where a queue sharing group includes queue managers that are not at the current version, events will be generated for any command that is executed by means of CMDSCOPE on a queue manager that is at the current version, but not on those that are at a previous version. This happens even if the queue manager where the command is entered is at the previous version, although in such a case no context information is included in the event data.

Related concepts
Introduction to Programmable Command Formats
"Configuration events" on page 36
Configuration events are notifications that are generated when an object is created, changed, or deleted, and can also be generated by explicit requests.

Related reference
MQSET - Set object attributes

Refresh Object configuration event

The Refresh Object configuration event is different from the other configuration events, because it occurs only when explicitly requested.

The create, change, and delete events are generated by an MQSET call or by a command to change an object but the refresh object event occurs only when explicitly requested by the MQSC command, REFRESH QMGR, or its PCF equivalent.

The REFRESH QMGR command is different from all the other commands that generate configuration events. All the other commands apply to a particular object and generate a single configuration event for that object. The REFRESH QMGR command can produce many configuration event messages potentially representing every object definition stored by a queue manager. One event message is generated for each object that is selected.

The REFRESH QMGR command uses a combination of three selection criteria to filter the number of objects involved:

- Object Name
- Object Type
- Refresh Interval

If you specify none of the selection criteria on the REFRESH QMGR command, the default values are used for each selection criteria and a refresh configuration event message is generated for every object definition stored by the queue manager. This might cause unacceptable processing times and event message generation. Consider specifying some selection criteria.

The REFRESH QMGR command that generates the refresh events can be used in the following situations:

- When configuration data is wanted about all or some of the objects in a system regardless of whether the objects have been recently manipulated, for example, when configuration events are first enabled.

  Consider using several commands, each with a different selection of objects, but such that all are included.

- If there has been an error in the SYSTEM.ADMIN.CONFIG.EVENT queue. In this circumstance, no configuration event messages are generated for Create, Change, or Delete events. When the error on the queue has been corrected, the Refresh Queue Manager command can be used to request the generation of event messages, which were lost while there was an error in the queue. In this situation consider setting the refresh interval to the time for which the queue was unavailable.

Related concepts
"Configuration events" on page 36
Configuration events are notifications that are generated when an object is created, changed, or deleted, and can also be generated by explicit requests.

Related reference
REFRESH QMGR
Refresh Queue Manager

Command events

Command events are notifications that an MQSC, or PCF command has run successfully.

The event data contains the following information:

Origin information
comprises the queue manager from where the command was issued, the ID of the user that issued the command, and how the command was issued, for example by a console command.

Context information
a replica of the context information in the message data from the command message. If a command is not entered using a message, context information is omitted.

Context information is included in the event data only when the command was entered as a message on the SYSTEM.COMMAND.INPUT queue.

Command information
the type of command that was issued.

Command data

- for PCF commands, a replica of the command data

- for MQSC commands, the command text

The command data format does not necessarily match the format of the original command. For example, on distributed platforms the command data format is always in PCF format, even if the original request was an MQSC command.

Every command event message that is generated is placed on the command event queue, SYSTEM.ADMIN.COMMAND.EVENT.

Related reference
Command
"Event types" on page 8
Use this page to view the types of instrumentation event that a queue manager or channel instance can report

Command event generation

Use this page to view the situations that cause command events to be generated and to understand the circumstances in which command events are not generated

When command events are not generated

A command event message is generated in the following situations:

- When the CMDEV queue manager attribute is specified as ENABLED and an MQSC or PCF command runs successfully.

- When the CMDEV queue manager attribute is specified as NODISPLAY and any command runs successfully, with the exception of DISPLAY commands (MQSC), and Inquire commands (PCF).
- When you run the MQSC command, ALTER QMGR, or the PCF command, Change Queue Manager, and the CMDEV queue manager attribute meets either of the following conditions:
  - CMDEV is not specified as DISABLED after the change
  - CMDEV was not specified as DISABLED before the change

If a command runs against the command event queue, SYSTEM.ADMIN.COMMAND.EVENT, a command event is generated if the queue still exists and it is not put-inhibited.

When command events are not generated

A command event message is not generated in the following circumstances:

- When a command fails
- When a queue manager encounters an error trying to put a command event on the event queue, in which case the command runs regardless, but no event message is generated
- For the MQSC command REFRESH QMGR TYPE (EARLY)
- For the MQSC command START QMGR MQSC
- For the MQSC command SUSPEND QMGR, if the parameter LOG is specified
- For the MQSC command RESUME QMGR, if the parameter LOG is specified

Related concepts
"Command events" on page 40
Command events are notifications that an MQSC, or PCF command has run successfully.

Related reference
REFRESH QMGR
SUSPEND QMGR
RESUME QMGR
SUSPEND QMGR, RESUME QMGR and clusters

Command event usage

Use this page to view how you can use command events to generate an audit trail of the commands that have run

For example, if an object is changed unexpectedly, information regarding who made the alteration and when it was done can be stored. This can be particularly useful when configuration events are also enabled. If an MQSC or PCF command causes a command event and a configuration event to be generated, both event messages will share the same correlation identifier in their message descriptor.

If a command event message is generated, but cannot be put on the command event queue, for example if the command event queue has not been defined, the command for which the command event was generated still runs regardless.

Effects of CMDSCOPE

For commands where CMDSCOPE is used, the command event message or messages will be generated on the queue manager or queue managers where the command runs, not where the command is entered. However, all the origin and context information in the event data will relate to the original command as entered, even where the command using CMDSCOPE is one that has been generated by the source queue manager.

Related concepts
"Command events" on page 40

Command events are notifications that an MQSC, or PCF command has run successfully.

Use this page to view the situations that cause command events to be generated and to understand the circumstances in which command events are not generated

Related reference
The MQSC commands
PCF commands and responses in groups

Logger events

Logger events are notifications that a queue manager has started writing to a new log extent.

The event data contains the following information:

- The name of the current log extent.
- The name of the earliest log extent needed for restart recovery.
- The name of the earliest log extent needed for media recovery.
- The directory in which the log extents are located.

Every logger event message that is generated is placed on the logger event queue, SYSTEM.ADMIN.LOGGER.EVENT.

Related reference
Logger
Use this page to view the types of instrumentation event that a queue manager or channel instance can report

Logger event generation

Use this page to view the situations that cause logger events to be generated and to understand the circumstances in which logger events are not generated

A logger event message is generated in the following situations:

- When the LOGGEREV queue manager attribute is specified as ENABLED and the queue manager starts writing to a new log extent or, on IBM i, a journal receiver.
- When the LOGGEREV queue manager attribute is specified as ENABLED and the queue manager starts.
- When the LOGGEREV queue manager attribute is changed from DISABLED to ENABLED.

Tip: You can use the RESET QMGR MQSC command to request a queue manager to start writing to a new log extent.

When logger events are not generated

A logger event message is not generated in the following circumstances:

- When a queue manager is configured to use circular logging.

  In this case, the LOGGEREV queue manager attribute is set as DISABLED and cannot be altered.

- When a queue manager encounters an error trying to put a logger event on the event queue, in which case the action that caused the event completes, but no event message is generated.

Related concepts
Logger events are notifications that a queue manager has started writing to a new log extent.

Related reference
LoggerEvent (MQLONG)
RESET QMGR

Logger event usage

Use this page to view how you can use logger events to determine the log extents that are no longer required for queue manager restart, or media recovery.

You can archive superfluous log extents to a medium such as tape for disaster recovery before removing them from the active log directory. Regular removal of superfluous log extents keeps disk space usage to a minimum.

If the LOGGEREV queue manager attribute is enabled, but a logger event message cannot be put on the logger event queue, for example because the event queue has not been defined, the action that caused the event continues regardless.

Related concepts
"Logger events" on page 42
Logger events are notifications that a queue manager has started writing to a new log extent.

Related reference
LoggerEvent (MQLONG)
"Logger event generation" on page 42
Use this page to view the situations that cause logger events to be generated and to understand the circumstances in which logger events are not generated

Sample program to monitor the logger event queue

Use this page to view a sample C program that monitors the logger event queue for new event messages, reads those messages, and puts the contents of the message to stdout.

```
/*******************************************************************************/
/*                                                                           */
/* Program name: AMQSLOG0.C                                                   */
/*                                                                           */
/* Description:  Sample C program to monitor the logger event queue and output*/
/*               a message to stdout when a logger event occurs              */
/* <N_OCO_COPYRIGHT>                                                         */
/* Licensed Materials - Property of IBM                                      */
/*                                                                           */
/* 63H9336                                                                   */
/* (c) Copyright IBM Corp. 2005, 2023. All Rights Reserved.                  */
/*                                                                           */
/* US Government Users Restricted Rights - Use, duplication or               */
/* disclosure restricted by GSA ADP Schedule Contract with                   */
/* IBM Corp.                                                                 */
/* <NOC_COPYRIGHT>                                                           */
/*******************************************************************************/
/*                                                                           */
/* Function: AMQSLOG is a sample program which monitors the logger event      */
/* queue for new event messages, reads those messages, and puts the contents */
/* of the message to stdout.                                                 */
/*                                                                           */
/*******************************************************************************/
/*                                                                           */
/* AMQSLOG has 1 parameter - the queue manager name (optional, if not         */
/* specified then the default queue manager is implied)                      */
/*                                                                           */
/*******************************************************************************/

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

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

/*******************************************************************************/
/* Constants                                                                 */
/*******************************************************************************/

#define   MAX_MESSAGE_LENGTH   8000
```

```
typedef struct _ParmTableEntry
{
  MQLONG  ConstVal;
  PMQCHAR Desc;
} ParmTableEntry;

ParmTableEntry ParmTable[] =
{
  0                          ,"",
  MQCA_Q_MGR_NAME            ,"Queue Manager Name",
  MQCMD_LOGGER_EVENT         ,"Logger Event Command",
  MQRC_LOGGER_STATUS         ,"Logger Status",
  MQCACF_CURRENT_LOG_EXTENT_NAME,"Current Log Extent",
  MQCACF_RESTART_LOG_EXTENT_NAME,"Restart Log Extent",
  MQCACF_MEDIA_LOG_EXTENT_NAME  ,"Media Log Extent",
  MQCACF_LOG_PATH            ,"Log Path"};

/*****************************************************************************/
/* Function prototypes                                                       */
/*****************************************************************************/

static void ProcessPCF(MQHCONN    hConn,
                       MQHOBJ     hEventQueue,
                       PMQCHAR    pBuffer);

static PMQCHAR ParmToString(MQLONG Parameter);

/*********************************************************************/
/* Function: main                                                    */
/*********************************************************************/
int main(int argc, char * argv[])
{
  MQLONG    CompCode;
  MQLONG    Reason;
  MQHCONN   hConn = MQHC_UNUSABLE_HCONN;
  MQOD      ObjDesc = { MQOD_DEFAULT };
  MQCHAR    QMName[MQ_Q_MGR_NAME_LENGTH+1] = "";
  MQCHAR    LogEvQ[MQ_Q_NAME_LENGTH] = "SYSTEM.ADMIN.LOGGER.EVENT";
  MQHOBJ    hEventQueue;
  PMQCHAR   pBuffer = NULL;

  printf("\n/************************************/\n");
  printf("/* Sample Logger Event Monitor start */\n");
  printf("/************************************/\n");

  /*****************************************************************/
  /* Parse any command line options                                */
  /*****************************************************************/

  if (argc > 1)
     strncpy(QMName, argv[1], (size_t)MQ_Q_MGR_NAME_LENGTH);

  pBuffer = (char *)malloc(MAX_MESSAGE_LENGTH);
  if (!pBuffer)
  {
    printf("Can't allocate %d bytes\n",MAX_MESSAGE_LENGTH);
    goto MOD_EXIT;
  }

  /*****************************************************************/
  /* Connect to the specified (or default) queue manager           */
  /*****************************************************************/

  MQCONN(QMName,
         &hConn,
         &CompCode,
         &Reason);

  if (Reason != MQCC_OK)
  {
    printf("Error in call to MQCONN, Reason %d, CompCode %d\n", Reason,
    CompCode);
    goto MOD_EXIT;
  }

  /* Open the logger event queue for input  */

  strncpy(ObjDesc.ObjectQMgrName,QMName, MQ_Q_MGR_NAME_LENGTH);
  strncpy(ObjDesc.ObjectName, LogEvQ, MQ_Q_NAME_LENGTH);

  MQOPEN(  hConn,
           &ObjDesc,
```

```
              MQOO_INPUT_EXCLUSIVE,
          &hEventQueue,
          &CompCode,
          &Reason);
  if (Reason)
  {
    printf("MQOPEN failed for queue manager %.48s Queue %.48s Reason: %d\n",
                                       ObjDesc.ObjectQMgrName,
                                ObjDesc.ObjectName,
                                Reason);
    goto MOD_EXIT;
  }
  else
  {
    ProcessPCF(hConn, hEventQueue, pBuffer);
  }

  MOD_EXIT:

  if (pBuffer != NULL) {
    free(pBuffer);
  }

  /*******************************************************************/
  /* Disconnect                                                      */
  /*******************************************************************/
  if (hConn != MQHC_UNUSABLE_HCONN) {
      MQDISC(&hConn, &CompCode, &Reason);
  }

  return 0;
}

/***************************************************************************/
/* Function: ProcessPCF                                                    */
/***************************************************************************/
/*                                                                         */
/* Input Parameters:  Handle to queue manager connection                   */
/*                    Handle to the opened logger event queue object       */
/*                    Pointer to a memory buffer to store the incoming PCF msg*/
/*                                                                         */
/* Output Parameters: None                                                 */
/*                                                                         */
/* Logic: Wait for messages to appear on the logger event queue and display */
/* their contents.                                                         */
/*                                                                         */
/***************************************************************************/

static void ProcessPCF(MQHCONN    hConn,
                       MQHOBJ     hEventQueue,
                       PMQCHAR    pBuffer)
{
  MQCFH   * pCfh;
  MQCFST  * pCfst;
  MQGMO     Gmo    = { MQGMO_DEFAULT };
  MQMD      Mqmd   = { MQMD_DEFAULT };
  PMQCHAR   pPCFCmd;
  MQLONG    Reason  = 0;
  MQLONG    CompCode;
  MQLONG    MsgLen;
  PMQCHAR   Parm = NULL;
                                          /* Set timeout value            */
  Gmo.Options      |= MQGMO_WAIT;
  Gmo.Options |= MQGMO_CONVERT;
  Gmo.WaitInterval = MQWI_UNLIMITED;
  /*******************************************************************/
  /* Process response Queue                                          */
  /*******************************************************************/
  while (Reason == MQCC_OK)
  {
    memcpy(&Mqmd.MsgId;    , MQMI_NONE, sizeof(Mqmd.MsgId));
    memset(&Mqmd.CorrelId, 0, sizeof(Mqmd.CorrelId));

    MQGET( hConn,
           hEventQueue,
          &Mqmd,
          &Gmo,
           MAX_MESSAGE_LENGTH,
           pBuffer,
          &MsgLen,
          &CompCode,
          &Reason);
```

```c
      if (Reason != MQCC_OK)
      {
        switch(Reason)
        {
          case MQRC_NO_MSG_AVAILABLE:
              printf("Timed out");
              break;

          default:
              printf("MQGET failed RC(%d)\n", Reason);
              break;
        }
        goto MOD_EXIT;
      }

      /*******************************************************************/
      /* Only expect PCF event messages on this queue                  */
      /*******************************************************************/
      if (memcmp(Mqmd.Format, MQFMT_EVENT, sizeof(Mqmd.Format)))
      {
       printf("Unexpected message format '%8.8s' received\n",Mqmd.Format);
       continue;
      }


      /*******************************************************************/
      /* Build the output by parsing the received PCF message, first the */
      /* header, then each of the parameters                           */
      /*******************************************************************/

      pCfh = (MQCFH *)pBuffer;

      if (pCfh -> Reason)
      {
       printf("-------------------------------------------------------------------\n");
       printf("Event Message Received\n");

       Parm = ParmToString(pCfh->Command);
       if (Parm != NULL) {
         printf("Command  :%s \n",Parm);
       }
       else
       {
         printf("Command  :%d \n",pCfh->Command);
       }

       printf("CompCode :%d\n"    ,pCfh->CompCode);

       Parm = ParmToString(pCfh->Reason);
       if (Parm != NULL) {
         printf("Reason    :%s \n",Parm);
       }
       else
       {
         printf("Reason    :%d \n",pCfh->Reason);
       }
      }

      pPCFCmd  = (char *)  (pCfh+1);
      printf("-------------------------------------------------------------------\n");
      while(pCfh -> ParameterCount--)
      {
        pCfst = (MQCFST *) pPCFCmd;
        switch(pCfst -> Type)
        {
          case MQCFT_STRING:
              Parm = ParmToString(pCfst -> Parameter);
              if (Parm != NULL) {
                printf("%-32s",Parm);
              }
              else
              {
                printf("%-32d",pCfst -> Parameter);
              }

              fwrite( pCfst -> String, pCfst -> StringLength, 1, stdout);
              pPCFCmd += pCfst -> StrucLength;
              break;
          default:
              printf("Unrecoginised datatype %d returned\n",pCfst->Type);
              goto MOD_EXIT;
        }
```

```
      putchar('\n');
    }
    printf("--------------------------------------------------------------\n");
  }
MOD_EXIT:

 return;
}


/*************************************************************************/
/* Function: ParmToString                                                */
/*************************************************************************/
/*                                                                       */
/* Input Parameters:   Parameter for which to get string description     */
/*                                                                       */
/* Output Parameters: None                                               */
/*                                                                       */
/* Logic: Takes a parameter as input and returns a pointer to a string   */
/* description for that parameter, or NULL if the parameter does not     */
/* have an associated string description                                 */
/*************************************************************************/

static PMQCHAR ParmToString(MQLONG Parameter){
  long i;
  for (i=0 ; i< sizeof(ParmTable)/sizeof(ParmTableEntry); i++)
  {
    if (ParmTable[i].ConstVal == Parameter ParmTable[i].Desc)
      return ParmTable[i].Desc;
  }
  return NULL;
}
```

Sample output

This application produces the following form of output:

```
/***********************************/
/* Sample Logger Event Monitor start */
/***********************************/
----------------------------------------------------------------
Event Message Received
Command  :Logger Event Command
CompCode :0
Reason   :Logger Status
----------------------------------------------------------------
Queue Manager Name              CSIM

Current Log Extent              AMQA000001
Restart Log Extent              AMQA000001
Media Log Extent                AMQA000001
Log Path                        QMCSIM
----------------------------------------------------------------
```

Related concepts

"Logger event usage" on page 43
Use this page to view how you can use logger events to determine the log extents that are no longer required for queue manager restart, or media recovery.

"Command event usage" on page 41
Use this page to view how you can use command events to generate an audit trail of the commands that have run

Related reference

"Logger event generation" on page 42

Use this page to view the situations that cause logger events to be generated and to understand the circumstances in which logger events are not generated

Sample program to monitor instrumentation events

Use this page to view a sample C program for monitoring instrumentation events

This sample program is not part of any IBM WebSphere MQ product and is therefore not supplied as an actual physical item. The example is incomplete in that it does not enumerate all the possible outcomes of specified actions. However, you can use this sample as a basis for your own programs that use events, in particular, the PCF formats used in event messages. However, you need to modify this program before running it on your own systems.

```
/**********************************************************************/
/*                                                                  */
/* Program name: EVMON                                              */
/*                                                                  */
/* Description: C program that acts as an event monitor            */
/*                                                                  */
/*                                                                  */
/**********************************************************************/
/*                                                                  */
/* Function:                                                        */
/*                                                                  */
/*                                                                  */
/*    EVMON is a C program that acts as an event monitor - reads an */
/*    event queue and tells you if anything appears on it          */
/*                                                                  */
/*    Its first parameter is the queue manager name, the second is  */
/*    the event queue name. If these are not supplied it uses the  */
/*    defaults.                                                     */
/*                                                                  */
/**********************************************************************/
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef min
  #define min(a,b)          (((a) < (b)) ? (a) : (b))
#endif

/**********************************************************************/
/* includes for MQI                                                 */
/**********************************************************************/
#include <cmqc.h>
#include <cmqcfc.h>
void printfmqcfst(MQCFST* pmqcfst);
void printfmqcfin(MQCFIN* pmqcfst);
void printreas(MQLONG reason);

 #define PRINTREAS(param)                                           \
    case param:                                                    \
      printf("Reason = %s\n",#param);                              \
      break;


/**********************************************************************/
/* global variable                                                  */
/**********************************************************************/
MQCFH    *evtmsg;                       /* evtmsg message buffer       */

int main(int argc, char **argv)
{
  /******************************************************************/
  /* declare variables                                            */
  /******************************************************************/
  int  i;                              /* auxiliary counter          */
  /******************************************************************/
  /* Declare MQI structures needed                                */
  /******************************************************************/
  MQOD    od = {MQOD_DEFAULT};        /* Object Descriptor          */
  MQMD    md = {MQMD_DEFAULT};        /* Message Descriptor         */
  MQGMO  gmo = {MQGMO_DEFAULT};       /* get message options        */
  /******************************************************************/
  /* note, uses defaults where it can                             */
```

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

MQHCONN  Hcon;                       /* connection handle          */
MQHOBJ   Hobj;                       /* object handle              */
MQLONG   O_options;                  /* MQOPEN options             */
MQLONG   C_options;                  /* MQCLOSE options            */
MQLONG   CompCode;                   /* completion code            */
MQLONG   OpenCode;                   /* MQOPEN completion code     */
MQLONG   Reason;                     /* reason code                */
MQLONG   CReason;                    /* reason code for MQCONN     */
MQLONG   buflen;                     /* buffer length              */
MQLONG   evtmsglen;                  /* message length received    */
MQCHAR   command[1100];              /* call command string ...    */
MQCHAR   p1[600];                    /* ApplId insert              */
MQCHAR   p2[900];                    /* evtmsg insert              */
MQCHAR   p3[600];                    /* Environment insert         */
MQLONG   mytype;                     /* saved application type     */
char     QMName[50];                 /* queue manager name         */
MQCFST   *paras;                     /* the parameters             */
int      counter;                    /* loop counter               */
time_t   ltime;

/********************************************************************/
/* Connect to queue manager                                         */
/********************************************************************/
QMName[0] = 0;                       /* default queue manager      */
if (argc > 1)
  strcpy(QMName, argv[1]);
MQCONN(QMName,                       /* queue manager              */
       &Hcon,                        /* connection handle          */
       &CompCode,                    /* completion code            */
       &CReason);                    /* reason code                */


/********************************************************************/
/* Initialize object descriptor for subject queue                   */
/********************************************************************/
strcpy(od.ObjectName, "SYSTEM.ADMIN.QMGR.EVENT");
if (argc > 2)
  strcpy(od.ObjectName, argv[2]);

/********************************************************************/
/* Open the event queue for input; exclusive or shared.  Use of    */
/* the queue is controlled by the queue definition here            */
/********************************************************************/

O_options = MQOO_INPUT_AS_Q_DEF     /* open queue for input       */
     + MQOO_FAIL_IF_QUIESCING       /* but not if qmgr stopping   */
     + MQOO_BROWSE;
MQOPEN(Hcon,                         /* connection handle          */
       &od,                          /* object descriptor for queue*/
       O_options,                    /* open options               */
       &Hobj,                        /* object handle              */
       &CompCode,                    /* completion code            */
       &Reason);                     /* reason code                */

/********************************************************************/
/*   Get messages from the message queue                            */
/********************************************************************/
while (CompCode != MQCC_FAILED)
{
  /****************************************************************/
  /* I don't know how big this message is so just get the         */
  /* descriptor first                                             */
  /****************************************************************/
  gmo.Options = MQGMO_WAIT + MQGMO_LOCK
     + MQGMO_BROWSE_FIRST + MQGMO_ACCEPT_TRUNCATED_MSG;
                                     /* wait for new messages      */
  gmo.WaitInterval = MQWI_UNLIMITED;/* no time limit               */
  buflen = 0;                        /* amount of message to get   */

  /****************************************************************/
  /* clear selectors to get messages in sequence                  */
  /****************************************************************/
  memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
  memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));

  /****************************************************************/
  /* wait for event message                                       */
  /****************************************************************/
  printf("...>\n");
```

```
          MQGET(Hcon,                        /* connection handle       */
                Hobj,                         /* object handle           */
                &md,                      /* message descriptor      */
                &gmo,                     /* get message options     */
                buflen,                       /* buffer length           */
                evtmsg,                       /* evtmsg message buffer   */
                &evtmsglen,               /* message length          */
                &CompCode,                /* completion code         */
                &Reason);                 /* reason code             */


          /*****************************************************************/
          /* report reason, if any                                       */
          /*****************************************************************/
          if (Reason != MQRC_NONE && Reason != MQRC_TRUNCATED_MSG_ACCEPTED)
          {
            printf("MQGET ==> %ld\n", Reason);
          }
          else
          {
            gmo.Options = MQGMO_NO_WAIT + MQGMO_MSG_UNDER_CURSOR;
            buflen = evtmsglen;               /* amount of message to get   */
            evtmsg = malloc(buflen);
            if (evtmsg != NULL)
            {
              /*************************************************************/
              /* clear selectors to get messages in sequence             */
              /*************************************************************/
              memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
              memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));

              /*************************************************************/
              /* get the event message                                   */
              /*************************************************************/
              printf("...>\n");
              MQGET(Hcon,                      /* connection handle       */
                    Hobj,                      /* object handle           */
                    &md,                   /* message descriptor      */
                    &gmo,                  /* get message options     */
                    buflen,                    /* buffer length           */
                    evtmsg,                    /* evtmsg message buffer   */
                    &evtmsglen,            /* message length          */
                    &CompCode,             /* completion code         */
                    &Reason);              /* reason code             */

              /*************************************************************/
              /* report reason, if any                                   */
              /*************************************************************/
              if (Reason != MQRC_NONE)
              {
                printf("MQGET ==> %ld\n", Reason);
              }
            }
            else
            {
              CompCode = MQCC_FAILED;
            }
          }
        }


        /*****************************************************************/
        /* . . . process each message received                         */
        /*****************************************************************/

        if (CompCode != MQCC_FAILED)
        {
          /*************************************************************/
          /* announce a message                                      */
          /*************************************************************/
          printf("\a\a\a\a\a\a");
          time(&ltime);
          printf(ctime(&ltime));

          if (evtmsglen != buflen)
            printf("DataLength = %ld?\n", evtmsglen);
          else
          {
            /*********************************************************/
            /* right let's look at the data                        */
            /*********************************************************/
            if (evtmsg->Type != MQCFT_EVENT)
```

```c
   {
     printf("Something's wrong this isn't an event message,"
            " its type is %ld\n",evtmsg->Type);
   }
   else
   {
     if (evtmsg->Command == MQCMD_Q_MGR_EVENT)
     {
       printf("Queue Manager event: ");
     }
     else
       if (evtmsg->Command == MQCMD_CHANNEL_EVENT)
       {
         printf("Channel event: ");
       }
       else
```

⋮

```c
       {
         printf("Unknown Event message, %ld.",
                evtmsg->Command);
       }

     if      (evtmsg->CompCode == MQCC_OK)
       printf("CompCode(OK)\n");
     else if (evtmsg->CompCode == MQCC_WARNING)
       printf("CompCode(WARNING)\n");
     else if (evtmsg->CompCode == MQCC_FAILED)
       printf("CompCode(FAILED)\n");
     else
       printf("* CompCode wrong * (%ld)\n",
              evtmsg->CompCode);

     if (evtmsg->StrucLength != MQCFH_STRUC_LENGTH)
     {
       printf("it's the wrong length, %ld\n",evtmsg->StrucLength);
     }

     if (evtmsg->Version != MQCFH_VERSION_1)
     {
       printf("it's the wrong version, %ld\n",evtmsg->Version);
     }

     if (evtmsg->MsgSeqNumber != 1)
     {
       printf("it's the wrong sequence number, %ld\n",
              evtmsg->MsgSeqNumber);
     }

     if (evtmsg->Control != MQCFC_LAST)
     {
       printf("it's the wrong control option, %ld\n",
              evtmsg->Control);
     }

     printreas(evtmsg->Reason);
     printf("parameter count is %ld\n", evtmsg->ParameterCount);
     /**********************************************************/
     /* get a pointer to the start of the parameters          */
     /**********************************************************/

     paras = (MQCFST *)(evtmsg + 1);
     counter = 1;
     while (counter <= evtmsg->ParameterCount)
     {
       switch (paras->Type)
       {
         case MQCFT_STRING:
           printfmqcfst(paras);
           paras = (MQCFST *)((char *)paras
                                + paras->StrucLength);
           break;
         case MQCFT_INTEGER:
           printfmqcfin((MQCFIN*)paras);
           paras = (MQCFST *)((char *)paras
                                + paras->StrucLength);
           break;
         default:
           printf("unknown parameter type, %ld\n",
                  paras->Type);
```

```
                       counter = evtmsg->ParameterCount;
                       break;
                   }
                   counter++;
                }
            }
        }   /* end evtmsg action              */
        free(evtmsg);
        evtmsg = NULL;
    }      /* end process for successful GET */
}          /* end message processing loop    */

  /********************************************************************/
  /* close the event queue - if it was opened                        */
  /********************************************************************/
  if (OpenCode != MQCC_FAILED)
  {
    C_options = 0;                    /* no close options            */
    MQCLOSE(Hcon,                     /* connection handle           */
            &Hobj,              /* object handle                     */
            C_options,
            &CompCode,            /* completion code                 */
            &Reason);             /* reason code                     */
  /********************************************************************/
  /* Disconnect from queue manager (unless previously connected)     */
  /********************************************************************/
  if (CReason != MQRC_ALREADY_CONNECTED)
  {
    MQDISC(&Hcon,                     /* connection handle           */
            &CompCode,            /* completion code                 */
            &Reason);             /* reason code                     */


  /********************************************************************/
  /*                                                                 */
  /* END OF EVMON                                                    */
  /*                                                                 */
  /********************************************************************/
  }

#define PRINTPARAM(param)                                             \
   case param:                                                        \
     {                                                                \
       char *p = #param;                                              \
     strncpy(thestring,pmqcfst->String,min(sizeof(thestring),       \
             pmqcfst->StringLength));                                 \
     printf("%s %s\n",p,thestring);                                   \
     }                                                                \
     break;

#define PRINTAT(param)                                                \
   case param:                                                        \
     printf("MQIA_APPL_TYPE = %s\n",#param);                          \
     break;


void printfmqcfst(MQCFST* pmqcfst)
{
  char thestring[100];

  switch (pmqcfst->Parameter)
  {
    PRINTPARAM(MQCA_BASE_Q_NAME)
    PRINTPARAM(MQCA_PROCESS_NAME)
    PRINTPARAM(MQCA_Q_MGR_NAME)
    PRINTPARAM(MQCA_Q_NAME)
    PRINTPARAM(MQCA_XMIT_Q_NAME)
    PRINTPARAM(MQCACF_APPL_NAME)
    :
    default:
     printf("Invalid parameter, %ld\n",pmqcfst->Parameter);
     break;
  }
}



void printfmqcfin(MQCFIN* pmqcfst)
{
  switch (pmqcfst->Parameter)
  {
```

```c
          case MQIA_APPL_TYPE:
            switch (pmqcfst->Value)
            {
              PRINTAT(MQAT_UNKNOWN)
              PRINTAT(MQAT_OS2)
              PRINTAT(MQAT_DOS)
              PRINTAT(MQAT_UNIX)
              PRINTAT(MQAT_QMGR)
              PRINTAT(MQAT_OS400)
              PRINTAT(MQAT_WINDOWS)
              PRINTAT(MQAT_CICS_VSE)
              PRINTAT(MQAT_VMS)
              PRINTAT(MQAT_GUARDIAN)
              PRINTAT(MQAT_VOS)
            }
            break;
          case MQIA_Q_TYPE:
            if (pmqcfst->Value == MQQT_ALIAS)
            {
              printf("MQIA_Q_TYPE is MQQT_ALIAS\n");
            }
            else
              :
              :
{
              if (pmqcfst->Value == MQQT_REMOTE)
              {
                printf("MQIA_Q_TYPE is MQQT_REMOTE\n");
                if (evtmsg->Reason == MQRC_ALIAS_BASE_Q_TYPE_ERROR)
                {
                  printf("but remote is not valid here\n");
                }
              }
              else
              {
                printf("MQIA_Q_TYPE is wrong, %ld\n",pmqcfst->Value);
              }
            }
            break;


            case MQIACF_REASON_QUALIFIER:
            printf("MQIACF_REASON_QUALIFIER %ld\n",pmqcfst->Value);
            break;

          case MQIACF_ERROR_IDENTIFIER:
            printf("MQIACF_ERROR_INDENTIFIER %ld (X'%lX')\n",
                   pmqcfst->Value,pmqcfst->Value);
            break;

          case MQIACF_AUX_ERROR_DATA_INT_1:
            printf("MQIACF_AUX_ERROR_DATA_INT_1 %ld (X'%lX')\n",
                   pmqcfst->Value,pmqcfst->Value);
            break;

          case MQIACF_AUX_ERROR_DATA_INT_2:
            printf("MQIACF_AUX_ERROR_DATA_INT_2 %ld (X'%lX')\n",
                   pmqcfst->Value,pmqcfst->Value);
            break;
      :
default :
            printf("Invalid parameter, %ld\n",pmqcfst->Parameter);
            break;
  }
}

  void printreas(MQLONG reason)
{
  switch (reason)
  {
    PRINTREAS(MQRCCF_CFH_TYPE_ERROR)
    PRINTREAS(MQRCCF_CFH_LENGTH_ERROR)
    PRINTREAS(MQRCCF_CFH_VERSION_ERROR)
    PRINTREAS(MQRCCF_CFH_MSG_SEQ_NUMBER_ERR)
     :
    PRINTREAS(MQRC_NO_MSG_LOCKED)
    PRINTREAS(MQRC_CONNECTION_NOT_AUTHORIZED)
    PRINTREAS(MQRC_MSG_TOO_BIG_FOR_CHANNEL)
    PRINTREAS(MQRC_CALL_IN_PROGRESS)
    default:
      printf("It's an unknown reason, %ld\n",
             reason);
      break;
```

```
        }
    }
}
```

Related concepts

"Instrumentation events" on page 6
An instrumentation event is a logical combination of conditions that a queue manager or channel instance detects and puts a special message, called an *event message*, on an event queue.

"Event monitoring" on page 5
Event monitoring is the process of detecting occurrences of *instrumentation events* in a queue manager network. An instrumentation event is a logical combination of events that is detected by a queue manager or channel instance. Such an event causes the queue manager or channel instance to put a special message, called an *event message*, on an event queue.

Related reference

C programming

"Sample program to monitor the logger event queue" on page 43
Use this page to view a sample C program that monitors the logger event queue for new event messages, reads those messages, and puts the contents of the message to stdout.

Message monitoring

Message monitoring is the process of identifying the route a message has taken through a queue manager network. By identifying the types of activities, and the sequence of activities performed on behalf of a message, the message route can be determined.

As a message passes through a queue manager network, various processes perform activities on behalf of the message. Use one of the following techniques to determine a message route:

- The IBM WebSphere MQ display route application (dspmqrte)
- Activity recording
- Trace-route messaging

These techniques all generate special messages that contain information about the activities performed on the message as it passed through a queue manager network. Use the information returned in these special messages to achieve the following objectives:

- Record message activity.
- Determine the last known location of a message.
- Detect routing problems in your queue manager network.
- Assist in determining the causes of routing problems in your queue manager network.
- Confirm that your queue manager network is running correctly.
- Familiarize yourself with the running of your queue manager network.
- Trace published messages.

Related concepts

Types of message

Activities and operations

Activities are discrete actions that an application performs on behalf of a message. Activities consist of operations, which are single pieces of work that an application performs.

The following actions are examples of activities:

- A message channel agent (MCA) sends a message from a transmission queue down a channel
- An MCA receives a message from a channel and puts it on its target queue

- An application getting a message from a queue, and putting a reply message in response.
- The WebSphere MQ publish/subscribe engine processes a message.

Activities consist of one or more *operations*. Operations are single pieces of work that an application performs. For example, the activity of an MCA sending a message from a transmission queue down a channel consists of the following operations:

1. Getting a message from a transmission queue (a *Get* operation).
2. Sending the message down a channel (a *Send* operation).

In a publish/subscribe network, the activity of the WebSphere MQ publish/subscribe engine processing a message can consist of the following multiple operations:

1. Putting a message to a topic string (a *Put* operation).
2. Zero or more operations for each of the subscribers that are considered for receipt of the message (a *Publish* operation, a *Discarded Publish* operation or an *Excluded Publish* operation).

Information from activities

You can identify the sequence of activities performed on a message by recording information as the message is routed through a queue manager network. You can determine the route of a message through the queue manager network from the sequence of activities performed on the message, and can obtain the following information:

The last known location of a message
If a message does not reach its intended destination, you can determine the last known location of the message from a complete or partial message route.

Configuration issues with a queue manager network
When studying the route of a message through a queue manager network, you might see that the message has not gone where expected. There are many reasons why this can occur, for example, if a channel is inactive, the message might take an alternative route.

For a publish/subscribe application, you can also determine the route of a message being published to a topic and any messages that flow in a queue manager network as a result of being published to subscribers.

In such situations, a system administrator can determine whether there are any problems in the queue manager network, and if appropriate, correct them.

Message routes

Depending on your reason for determining a message route, you can use the following general approaches:

Using activity information recorded for a trace-route message
Trace-route messages record activity information for a specific purpose. You can use them to determine configuration issues with a queue manager network, or to determine the last known location of a message. If a trace-route message is generated to determine the last known location of a message that did not reach its intended destination, it can mimic the original message. This gives the trace-route message the greatest chance of following the route taken by the original message.

The WebSphere MQ display route application can generate trace-route messages.

Using activity information recorded for the original message
You can enable any message for activity recording and have activity information recorded on its behalf. If a message does not reach its intended destination, you can use the recorded activity information to determine the last known location of the message. By using activity information from the original message, the most accurate possible message route can be determined, leading to the last known location. To use this approach, the original message must be enabled for activity recording.

Warning: Avoid enabling all messages in a queue manager network for activity recording. Messages enabled for activity recording can have many activity reports generated on their behalf. If every

message in a queue manager network is enabled for activity recording, the queue manager network traffic can increase to an unacceptable level.

Related concepts

"Message monitoring" on page 54
Message monitoring is the process of identifying the route a message has taken through a queue manager network. By identifying the types of activities, and the sequence of activities performed on behalf of a message, the message route can be determined.

"Message route techniques" on page 56
Activity recording and trace-route messaging are techniques that allow you to record activity information for a message as it is routed through a queue manager network.

"Trace-route messaging" on page 62
Trace-route messaging is a technique that uses *trace-route messages* to record activity information for a message. Trace-route messaging involves sending a trace-route message into a queue manager network.

Related tasks

Writing your own message channel agents

Message route techniques

Activity recording and trace-route messaging are techniques that allow you to record activity information for a message as it is routed through a queue manager network.

Activity recording

If a message has the appropriate report option specified, it requests that applications generate *activity reports* as it is routed through a queue manager network. When an application performs an activity on behalf of a message, an activity report can be generated, and delivered to an appropriate location. An activity report contains information about the activity that was performed on the message.

The activity information collected using activity reports must be arranged in order before a message route can be determined.

Trace-route messaging

Trace-route messaging is a technique that involves sending a *trace-route message* into a queue manager network. When an application performs an activity on behalf of the trace-route message, activity information can be accumulated in the message data of the trace-route message, or activity reports can be generated. If activity information is accumulated in the message data of the trace-route message, when it reaches its target queue a trace-route reply message containing all the information from the trace-route message can be generated and delivered to an appropriate location.

Because a trace-route message is dedicated to recording the sequence of activities performed on its behalf, there are more processing options available compared with normal messages that request activity reports.

Comparison of activity recording and trace-route messaging

Both activity recording and trace-route messaging can provide activity information to determine the route a message has taken through a queue manager network. Both methods have their own advantages.

Benefit	Activity recording	Trace-route messaging
Can determine the last known location of a message	Yes	Yes
Can determine configuration issues with a queue manager network	Yes	Yes
Can be requested by any message (is not restricted to use with trace-route messages)	Yes	No
Message data is left unmodified	Yes	No

Benefit	Activity recording	Trace-route messaging
Message processed normally	Yes	No
Activity information can be accumulated in the message data	No	Yes
Optional message delivery to target queue	No	Yes
If a message is caught in an infinite loop, it can be detected and dealt with	No	Yes
Activity information can be put in order reliably	No	Yes
Application provided to display the activity information	No	Yes

Message route completeness

In some cases it is not possible to identify the full sequence of activities performed on behalf of a message, so only a partial message route can be determined. The completeness of a message route is directly influenced by the queue manager network that the messages are routed through. The completeness of a message route depends on the level of the queue managers in the queue manager network, as follows:

Queue managers at WebSphere MQ Version 6.0 and subsequent releases
   MCAs and user-written applications connected to queue managers at WebSphere MQ Version 6.0 or subsequent releases can record information related to the activities performed on behalf of a message. The recording of activity information is controlled by the queue manager attributes ACTIVREC and ROUTEREC. If a queue manager network consists of queue managers at WebSphere MQ Version 6.0 or subsequent releases only, complete message routes can be determined.

WebSphere MQ queue managers before Version 6.0
   Applications connected to WebSphere MQ queue managers before Version 6.0 **do not** record the activities that they have performed on behalf of a message. If a queue manager network contains any WebSphere MQ queue manager prior to Version 6.0, only a partial message route can be determined.

How activity information is stored

WebSphere MQ stores activity information in activity reports, trace-route messages, or trace-route reply messages. In each case the information is stored in a structure called the *Activity* PCF group. A trace-route message or trace-route reply message can contain many Activity PCF groups, depending on the number of activities performed on the message. Activity reports contain one Activity PCF group because a separate activity report is generated for every recorded activity.

With trace-route messaging, additional information can be recorded. This additional information is stored in a structure called the *TraceRoute* PCF group. The TraceRoute PCF group contains a number of PCF structures that are used to store additional activity information, and to specify options that determine how the trace-route message is handled as it is routed through a queue manager network.

Related concepts
"Activity recording" on page 58
Activity recording is a technique for determining the routes that messages take through a queue manager network. To determine the route that a message has taken, the activities performed on behalf of the message are recorded.

"Trace-route messaging" on page 62
Trace-route messaging is a technique that uses *trace-route messages* to record activity information for a message. Trace-route messaging involves sending a trace-route message into a queue manager network.

Related reference
"The TraceRoute PCF group" on page 68

Attributes in the *TraceRoute* PCF group control the behavior of a trace-route message. The *TraceRoute* PCF group is in the message data of every trace-route message.

"Activity report message data" on page 102
Use this page to view the parameters contained by the *Activity* PCF group in an activity report message. Some parameters are returned only when specific operations have been performed.

Activity recording

Activity recording is a technique for determining the routes that messages take through a queue manager network. To determine the route that a message has taken, the activities performed on behalf of the message are recorded.

When using activity recording, each activity performed on behalf of a message can be recorded in an activity report. An activity report is a type of report message. Each activity report contains information about the application that performed the activity on behalf of the message, when the activity took place, and information about the operations that were performed as part of the activity. Activity reports are typically delivered to a reply-to queue where they are collected together. By studying the activity reports related to a message, you can determine the route that the message took through the queue manager network.

Activity report usage

When messages are routed through a queue manager network, activity reports can be generated. You can use activity report information in the following ways:

Determine the last known location of a message
If a message that is enabled for activity recording does not reach its intended destination, activity reports generated for the message as it was routed through a queue manager network can be studied to determine the last known location of the message.

Determine configuration issues with a queue manager network
A number of messages enabled for activity recording can be sent into a queue manager network. By studying the activity reports related to each message it can become apparent that they have not taken the expected route. There are many reasons why this can occur, for example, a channel could have stopped, forcing the message to take an alternative route. In these situations, a system administrator can determine whether there are any problems in the queue manager network, and if there are, correct them.

Note: You can use activity recording in conjunction with trace-route messages by using the WebSphere MQ display route application.

Activity report format

Activity reports are PCF messages generated by applications that have performed an activity on behalf of a message. Activity reports are standard WebSphere MQ report messages containing a message descriptor and message data, as follows:

The message descriptor

- An MQMD structure

Message data

- An embedded PCF header (MQEPH)
- Activity report message data

Activity report message data consists of the *Activity* PCF group, and if generated for a trace-route message, the *TraceRoute* PCF group.

Related reference
MQMD - Message descriptor
MQEPH - Embedded PCF header

Controlling activity recording

Enable activity recording at the queue manager level. To enable an entire queue manager network, individually enable every queue manager in the network for activity recording. If you enable more queue managers, more activity reports are generated.

About this task

To generate activity reports for a message as it is routed through a queue manager: define the message to request activity reports; enable the queue manager for activity recording; and ensure that applications performing activities on the message are capable of generating activity reports.

If you do *not* want activity reports to be generated for a message as it is routed through a queue manager, *disable* the queue manager for activity recording.

Procedure

1. Request activity reports for a message

    a) In the message descriptor of the message, specify MQRO_ACTIVITY in the *Report* field.

    b) In the message descriptor of the message, specify the name of a reply-to queue in the *ReplyToQ* field.

    Warning: Avoid enabling all messages in a queue manager network for activity recording. Messages enabled for activity recording can have many activity reports generated on their behalf. If every message in a queue manager network is enabled for activity recording, the queue manager network traffic can increase to an unacceptable level.

2. Enable or disable the queue manager for activity recording.

    Use the MQSC command ALTER QMGR, specifying the parameter ACTIVREC, to change the value of the queue manager attribute. The value can be:

    MSG
    The queue manager is enabled for activity recording. Any activity reports generated are delivered to the reply-to queue specified in the message descriptor of the message. This is the default value.

    QUEUE
    The queue manager is enabled for activity recording. Any activity reports generated are delivered to the local system queue SYSTEM.ADMIN.ACTIVITY.QUEUE. The system queue can also be used to forward activity reports to a common queue.

    DISABLED
    The queue manager is disabled for activity recording. No activity reports are generated while in the scope of this queue manager.

    For example, to enable a queue manager for activity recording and specify that any activity reports generated are delivered to the local system queue SYSTEM.ADMIN.ACTIVITY.QUEUE, use the following MQSC command:

    ```
    ALTER QMGR ACTIVREC(QUEUE)
    ```

    Remember: When you modify the *ACTIVREC* queue manager attribute, a running MCA does not detect the change until the channel is restarted.

3. Ensure that your application uses the same algorithm as MCAs use to determine whether to generate an activity report for a message:

    a) Verify that the message has requested activity reports to be generated

    b) Verify that the queue manager where the message currently resides is enabled for activity recording

    c) Put the activity report on the queue determined by the *ACTIVREC* queue manager attribute

Setting up a common queue for activity reports

To determine the locations of the activity reports related to a specific message when the reports are delivered to the local system queue, it is more efficient to use a common queue on a single node

Before you begin
Set the ACTIVREC parameter to enable the queue manager for activity recording and to specify that any activity reports generated are delivered to the local system queue SYSTEM.ADMIN.ACTIVITY.QUEUE.

About this task
If a number of queue managers in a queue manager network are set to deliver activity reports to the local system queue, it can be time consuming to determine the locations of the activity reports related to a specific message. Alternatively, use a single node, which is a queue manager that hosts a common queue. All the queue managers in a queue manager network can deliver activity reports to this common queue. The benefit of using a common queue is that queue managers do not have to deliver activity reports to the reply-to queue specified in a message and, when determining the locations of the activity reports related to a message, you query one queue only.

To set up a common queue, perform the following steps:

Procedure

1. Select or define a queue manager as the single node
2. On the single node, select or define a queue for use as the common queue
3. On all queue managers where activity reports are to be delivered to the common queue, redefine the local system queue SYSTEM.ADMIN.ACTIVITY.QUEUE as a remote queue definition:
    a) Specify the name of the single node as the remote queue manager name
    b) Specify the name of the common queue as the remote queue name

Determining message route information

To determine a message route, obtain the information from the activity reports collected. Determine whether enough activity reports are on the reply-to queue to enable you to determine the required information and arrange the activity reports in order.

About this task
The order that activity reports are put on the reply-to queue does not necessarily correlate to the order in which the activities were performed. You must order activity reports manually, unless they are generated for a trace-route message, in which case you can use the WebSphere MQ display route application to order the activity reports.

Determine whether enough activity reports are on the reply-to queue for you to obtain the necessary information:

Procedure

1. Identify all related activity reports on the reply-to queue by comparing identifiers of the activity reports and the original message. Ensure you set the report option of the original message such that the activity reports can be correlated with the original message.
2. Order the identified activity reports from the reply-to queue.

    You can use the following parameters from the activity report:

    OperationType

        The types of operations performed might enable you to determine the activity report that was generated directly before, or after, the current activity report.

For example, an activity report details that an MCA sent a message from a transmission queue down a channel. The last operation detailed in the activity report has an *OperationType* of `send` and details that the message was sent using the channel, CH1, to the destination queue manager, QM1. This means that the next activity performed on the message will have occurred on queue manager, QM1, and that it will have begun with a `receive` operation from channel, CH1. By using this information you can identify the next activity report, providing it exists and has been acquired.

OperationDate and **OperationTime**

You can determine the general order of the activities from the dates and times of the operations in each activity report.

Warning: Unless every queue manager in the queue manager network has their system clocks synchronized, ordering by date and time does not guarantee that the activity reports are in the correct sequence. You must establish the order manually.

The order of the activity reports represents the route, or partial route, that the message took through the queue manager network.

3. Obtain the information you need from the activity information in the ordered activity reports.

If you have insufficient information about the message, you might be able to acquire further activity reports.

Retrieving further activity reports

To determine a message route, sufficient information must be available from the activity reports collected. If you retrieve the activity reports related to a message from the reply-to queue that the message specified, but you not have the necessary information, look for further activity reports.

About this task
To determine the locations of any further activity reports, perform the following steps:

Procedure

1. For any queue managers in the queue manager network that deliver activity reports to a common queue, retrieve activity reports from the common queue that have a *CorrelId* that matches the *MsgId* of the original message.

2. For any queue managers in the queue manager network that do not deliver activity reports to a common queue, retrieve activity reports as follows:

   a) Examine the existing activity reports to identify queue managers through which the message was routed.

   b) For these queue managers, identify the queue managers that are enabled for activity recording.

   c) For these queue managers, identify any that did not return activity reports to the specified reply-to queue.

   d) For each of the queue managers that you identify, check the system queue SYSTEM.ADMIN.ACTIVITY.QUEUE and retrieve any activity reports that have a *CorrelId* that matches the *MsgId* of the original message.

   e) If you find no activity reports on the system queue, check the queue manager dead letter queue, if one exists.

   An activity report can only be delivered to a dead letter queue if the report option, MQRO_DEAD_LETTER_Q, is set.

3. Arrange all the acquired activity reports in order.

   The order of the activity reports then represents the route, or partial route, that the message took.

4. Obtain the information you need from the activity information in the ordered activity reports.

   In some circumstances, recorded activity information cannot reach the specified reply-to queue, a common queue, or a system queue.

Circumstances where activity information is not acquired

To determine the complete sequence of activities performed on behalf of a message, information related to every activity must be acquired. If the information relating to any activity has not been recorded, or has not been acquired, you can determine only a partial sequence of activities.

Activity information is not recorded in the following circumstances:

- The message is processed by a WebSphere MQ queue manager earlier than Version 6.0.
- The message is processed by a queue manager that is not enabled for activity recording.
- The application that expected to process the message is not running.

Recorded activity information is unable to reach the specified reply-to queue in the following circumstances:

- There is no channel defined to route activity reports to the reply-to queue.
- The channel to route activity reports to the reply-to queue is not running.
- The remote queue definition to route activity reports back to the queue manager where the reply-to queue resides (the queue manager alias), is not defined.
- The user that generated the original message does not have open, or put, authority to the queue manager alias.
- The user that generated the original message does not have open, or put, authority to the reply-to queue.
- The reply-to queue is put inhibited.

Recorded activity information is unable to reach the system queue, or a common queue, in the following circumstances:

- If a common queue is to be used and there is no channel defined to route activity reports to the common queue.
- If a common queue is to be used and the channel to route activity reports to the common queue is not running.
- If a common queue is to be used and the system queue is incorrectly defined.
- The user that generated the original message does not have open, or put, authority to the system queue.
- The system queue is put inhibited.
- If a common queue is to be used and the user that generated the original message does not have open, or put, authority to the common queue.
- If a common queue is to be used and the common queue is put inhibited.

In these circumstances, providing the activity report does not have the report option MQRO_DISCARD_MSG specified, the activity report can be retrieved from a dead letter queue if one was defined on the queue manager where the activity report was rejected. An activity report will only have this report option specified if the original message, from which the activity report was generated, had both MQRO_PASS_DISCARD_AND_EXPIRY and MQRO_DISCARD_MSG specified in the Report field of the message descriptor.

Trace-route messaging

Trace-route messaging is a technique that uses *trace-route messages* to record activity information for a message. Trace-route messaging involves sending a trace-route message into a queue manager network.

As the trace-route message is routed through the queue manager network, activity information is recorded. This activity information includes information about the applications that performed the activities, when they were performed, and the operations that were performed as part of the activities. You can use the information recorded using trace-route messaging for the following purposes:

To determine the last known location of a message

If a message does not reach its intended destination, you can use the activity information recorded for a trace-route message to determine the last known location of the message. A trace-route message is sent into a queue manager network with the same target destination as the original message, intending that it follows the same route. Activity information can be accumulated in the message data of the trace-route message, or recorded using activity reports. To increase the probability that the trace-route message follows the same route as the original message, you can modify the trace-route message to mimic the original message.

To determine configuration issues with a queue manager network

Trace-route messages are sent into a queue manager network and activity information is recorded. By studying the activity information recorded for a trace-route message, it can become apparent that the trace-route message did not follow the expected route. There are many reasons why this can occur, for example, a channel might be inactive, forcing the message to take an alternative route. In these situations, a system administrator can determine whether there are any problems in the queue manager network, and if there are, correct them.

You can use the WebSphere MQ display route application to configure, generate, and put trace-route messages into a queue manager network.

Warning: If you put a trace-route message to a distribution list, the results are undefined.

Related concepts

"Trace-route message reference" on page 120
Use this page to obtain an overview of the trace-route message format. The trace-route message data includes parameters that describe the activities that the trace-route message has caused

How activity information is recorded

With trace-route messaging, you can record activity information in the message data of the trace-route message, or use activity reports. Alternatively, you can use both techniques.

Accumulating activity information in the message data of the trace-route message

As a trace-route message is routed through a queue manager network, information about the activities performed on behalf of the trace-route message can be accumulated in the message data of the trace-route message. The activity information is stored in *Activity* PCF groups. For every activity performed on behalf of the trace-route message, an *Activity* PCF group is written to the end of the PCF block in the message data of the trace-route message.

Additional activity information is recorded in trace-route messaging, in a PCF group called the *TraceRoute* PCF group. The additional activity information is stored in this PCF group, and can be used to help determine the sequence of recorded activities. This technique is controlled by the *Accumulate* parameter in the *TraceRoute* PCF group.

Recording activity information using activity reports

As a trace-route message is routed through a queue manager network, an activity report can be generated for every activity that was performed on behalf of the trace-route message. The activity information is stored in the *Activity* PCF group. For every activity performed on behalf of a trace-route message, an activity report is generated containing an *Activity* PCF group. Activity recording for trace-route messages works in the same way as for any other message.

Activity reports generated for trace-route messages contain additional activity information compared to the those generated for any other message. The additional information is returned in a *TraceRoute* PCF group. The information contained in the *TraceRoute* PCF group is accurate only from the time the activity report was generated. You can use the additional information to help determine the sequence of activities performed on behalf of the trace-route message.

Acquiring recorded activity information

When a trace-route message has reached its intended destination, or is discarded, the method that you use to acquire the activity information depends on how that information was recorded.

Before you begin

If you are unfamiliar with activity information, refer to "How activity information is recorded" on page 63.

About this task

Use the following methods to acquire the activity information after the trace-route message has reached its intended destination, or is discarded:

Procedure

- Retrieve the trace-route message.

  The *Deliver* parameter, in the *TraceRoute* PCF group, controls whether a trace-route message is placed on the target queue on arrival, or whether it is discarded. If the trace-route message is delivered to the target queue, you can retrieve the trace-route message from this queue. Then, you can use the WebSphere MQ display route application to display the activity information.

  To request that activity information is accumulated in the message data of a trace-route message, set the *Accumulate* parameter in the *TraceRoute* PCF group to MQROUTE_ACCUMULATE_IN_MSG.

- Use a trace-route reply message.

  When a trace-route message reaches its intended destination, or the trace-route message cannot be routed any further in a queue manager network, a trace-route reply message can be generated. A trace-route reply message contains a duplicate of all the activity information from the trace-route message, and is either delivered to a specified reply-to queue, or the system queue SYSTEM.ADMIN.TRACE.ROUTE.QUEUE. You can use the WebSphere MQ display route application to display the activity information.

  To request a trace-route reply message, set the *Accumulate* parameter in the *TraceRoute* PCF group to MQROUTE_ACCUMULATE_AND_REPLY.

- Use activity reports.

  If activity reports are generated for a trace-route message, you must locate the activity reports before you can acquire the activity information. Then, to determine the sequence of activities, you must order the activity reports.

Controlling trace-route messaging

Enable trace-route messaging at the queue manager level, so that applications in the scope of that queue manager can write activity information to a trace-route message. To enable an entire queue manager network, individually enable every queue manager in the network for trace-route messaging. If you enable more queue managers, more activity reports are generated.

Before you begin

If you are using activity reports to record activity information for a trace-route message, refer to "Controlling activity recording" on page 59.

About this task

To record activity information for a trace-route message as it is routed through a queue manager, perform the following steps:

Procedure

- Define how activity information is to be recorded for the trace-route message.

  Refer to "Generating and configuring a trace-route message" on page 67

- If you want to accumulate activity information in the trace-route message, ensure that the queue manager is enabled for trace-route messaging
- If you want to accumulate activity information in the trace-route message, ensure that applications performing activities on the trace-route message are capable of writing activity information to the message data of the trace-route message

Related concepts

"Generating and configuring a trace-route message" on page 67
A trace-route message comprises specific message descriptor and message data parts. To generate a trace-route message, either create the message manually or use the WebSphere MQ display route application.

Related tasks

"Controlling activity recording" on page 59
Enable activity recording at the queue manager level. To enable an entire queue manager network, individually enable every queue manager in the network for activity recording. If you enable more queue managers, more activity reports are generated.

Enabling queue managers for trace-route messaging

To control whether queue managers are enabled or disabled for trace-route messaging use the queue manager attribute ROUTEREC.

Use the MQSC command ALTER QMGR, specifying the parameter ROUTEREC to change the value of the queue manager attribute. The value can be:

MSG

The queue manager is enabled for trace-route messaging. Applications within the scope of the queue manager can write activity information to the trace-route message.

If the *Accumulate* parameter in the *TraceRoute* PCF group is set as MQROUTE_ACCUMULATE_AND_REPLY, and the next activity to be performed on the trace-route message:

- is a discard
- is a put to a local queue (target queue or dead-letter queue)
- will cause the total number of activities performed on the trace-route message to exceed the value of parameter the *MaxActivities*, in the *TraceRoute* PCF group .

a trace-route reply message is generated, and delivered to the reply-to queue specified in the message descriptor of the trace-route message.

QUEUE

The queue manager is enabled for trace-route messaging. Applications within the scope of the queue manager can write activity information to the trace-route message.

If the *Accumulate* parameter in the *TraceRoute* PCF group is set as MQROUTE_ACCUMULATE_AND_REPLY, and the next activity to be performed on the trace-route message:

- is a discard
- is a put to a local queue (target queue or dead-letter queue)
- will cause the total number of activities performed on the trace-route message to exceed the value of parameter the *MaxActivities*, in the *TraceRoute* PCF group .

a trace-route reply message is generated, and delivered to the local system queue SYSTEM.ADMIN.TRACE.ROUTE.QUEUE.

DISABLED

The queue manager is disabled for trace-route messaging. Activity information is not accumulated in the the trace-route message, however the *TraceRoute* PCF group can be updated while in the scope of this queue manager.

For example, to disable a queue manager for trace-route messaging, use the following MQSC command:

```
ALTER QMGR ROUTEREC(DISABLED)
```

Remember: When you modify the *ROUTEREC* queue manager attribute, a running MCA does not detect the change until the channel is restarted.

Enabling applications for trace-route messaging
To enable trace-route messaging for a user application, base your algorithm on the algorithm used by message channel agents (MCAs)

Before you begin
If you are not familiar with the format of a trace-route message, see .

About this task
Message channel agents (MCAs) are enabled for trace-route messaging. To enable a user application for trace-route messaging, use the following steps from the algorithm that MCAs use:

Procedure

1. Determine whether the message being processed is a trace-route message.

   If the message does not conform to the format of a trace-route message, the message is not processed as a trace-route message.

2. Determine whether activity information is to be recorded.

   If the detail level of the performed activity is not less than the level of detail specified by the *Detail* parameter, activity information is recorded under specific circumstances. This information is only recorded if the trace-route message requests accumulation, and the queue manager is enabled for trace-route messaging, or if the trace-route message requests an activity report and the queue manager is enabled for activity recording.

   - If activity information is to be recorded, increment the *RecordedActivities* parameter.

   - If activity information is not to be recorded, increment the *UnrecordedActivities* parameter.

3. Determine whether the total number of activities performed on the trace-route message exceeds the value of the *MaxActivities* parameter.

   The total number of activities is the sum of *RecordedActivities*, *UnrecordedActivities*, and *DiscontinuityCount*.

   If the total number of activities exceeds *MaxActivities*, reject the message with feedback MQFB_MAX_ACTIVITIES.

4. If value of *Accumulate* is set as MQROUTE_ACCUMULATE_IN_MSG or MQROUTE_ACCUMULATE_AND_REPLY, and the queue manager is enabled for trace-route messaging, write an Activity PCF group to the end of the PCF block in the message data of a trace-route message.

5. Deliver the trace-route message to a local queue.

   - If the parameter, *Deliver*, is specified as MQROUTE_DELIVER_NO, reject the trace-route message with feedback MQFB_NOT_DELIVERED.

   - If the parameter, *Deliver*, is specified as MQROUTE_DELIVER_YES, deliver the trace-route message to the local queue.

6. Generate a trace-route reply message if all the following conditions are true:

   - The trace-route message was delivered to a local queue or rejected

   - The value of the parameter, *Accumulate*, is MQROUTE_ACCUMULATE_AND_REPLY

   - The queue manager is enabled for trace-route messaging

The trace-route reply message is put on the queue determined by the ROUTEREC queue manager attribute.

7. If the trace-route message requested an activity report and the queue manager is enabled for activity recording, generate an activity report.

   The activity report is put on the queue determined by the ACTIVREC queue manager attribute.

Generating and configuring a trace-route message

A trace-route message comprises specific message descriptor and message data parts. To generate a trace-route message, either create the message manually or use the WebSphere MQ display route application.

A trace-route message consists of the following parts:

Message descriptor
An MQMD structure, with the *Format* field set to MQFMT_ADMIN or MQFMT_EMBEDDED_PCF.

Message data
One of the following combinations:

- A PCF header (MQCFH) and trace-route message data, if *Format* is set to MQFMT_ADMIN
- An embedded PCF header (MQEPH), trace-route message data, and additional user-specified message data, if *Format* is set to MQFMT_EMBEDDED_PCF

The trace-route message data consists of the *TraceRoute* PCF group and one or more *Activity* PCF groups.

Manual generation

When generating a trace-route message manually, an *Activity* PCF group is not required. *Activity* PCF groups are written to the message data of the trace-route message when an MCA or user-written application performs an activity on its behalf.

The WebSphere MQ display route application

Use the WebSphere MQ display route application, `dspmqrte`, to configure, generate and put a trace-route message into a queue manager network. Set the *Format* parameter in the message descriptor to MQFMT_ADMIN. You cannot add user data to the trace-route message generated by the WebSphere MQ display route application.

Restriction: `dspmqrte` cannot be issued on queue managers before WebSphere MQ Version 6.0 or on WebSphere MQ for z/OS queue managers. If you want the first queue manager the trace-route message is routed through to be a queue manager of this type, connect to the queue manager as a WebSphere MQ Version 6.0 or later client using the optional parameter `-c`.

Mimicking the original message

When using a trace-route message to determine the route another message has taken through a queue manager network, the more closely a trace-route message mimics the original message, the greater the chance that the trace-route message will follow the same route as the original message.

The following message characteristics can affect where a message is forwarded to within a queue manager network:

Priority
The priority can be specified in the message descriptor of the message.

Persistence
The persistence can be specified in the message descriptor of the message.

Expiration
The expiration can be specified in the message descriptor of the message.

Report options
Report options can be specified in the message descriptor of the message.

Message size

    To mimic the size of a message, additional data can be written to the message data of the message. For this purpose, additional message data can be meaningless.

    Tip: The WebSphere MQ display route application cannot specify message size.

Message data

    Some queue manager networks use content based routing to determine where messages are forwarded. In these cases the message data of the trace-route message needs to be written to mimic the message data of the original message.

    Tip: The WebSphere MQ display route application cannot specify message data.

The TraceRoute PCF group

Attributes in the *TraceRoute* PCF group control the behavior of a trace-route message. The *TraceRoute* PCF group is in the message data of every trace-route message.

The following table lists the parameters in the *TraceRoute* group that an MCA recognizes. Further parameters can be added if user-written applications are written to recognize them, as described in "Additional activity information" on page 73.

Table 12. TraceRoute PCF group	
Parameter	**Type**
TraceRoute	MQCFGR
Detail	MQCFIN
RecordedActivities	MQCFIN
UnrecordedActivities	MQCFIN
DiscontinuityCount	MQCFIN
MaxActivities	MQCFIN
Accumulate	MQCFIN
Forward	MQCFIN
Deliver	MQCFIN

Descriptions of each parameter in the *TraceRoute* PCF group follows:

Detail

    Specifies the detail level of activity information that is to be recorded. The value can be:

    MQROUTE_DETAIL_LOW

        Only activities performed by user application are recorded.

    MQROUTE_DETAIL_MEDIUM

        Activities specified in MQROUTE_DETAIL_LOW should be recorded. Additionally, activities performed by MCAs are recorded.

    MQROUTE_DETAIL_HIGH

        Activities specified in MQROUTE_DETAIL_LOW, and MQROUTE_DETAIL_MEDIUM should be recorded. MCAs do not record any further activity information at this level of detail. This option is only available to user applications that are to record further activity information. For example, if a user application determines the route a message takes by considering certain message characteristics, the information about the routing logic could be included with this level of detail.

RecordedActivities

    Specifies the number of recorded activities performed on behalf of the trace-route message. An activity is considered to be recorded if information about it has been written to the trace-route message, or if an activity report has been generated. For every recorded activity, *RecordedActivities* increments by one.

UnrecordedActivities
Specifies the number of unrecorded activities performed on behalf of the trace-route message. An activity is considered to be unrecorded if an application that is enabled for trace-route messaging neither accumulates, nor writes the related activity information to an activity report.

An activity performed on behalf of a trace-route message is unrecorded in the following circumstances:

- The detail level of the performed activity is less than the level of detail specified by the parameter *Detail*.
- The trace-route message requests an activity report but not accumulation, and the queue manager is not enabled for activity recording.
- The trace-route message requests accumulation but not an activity report, and the queue manager is not enabled for trace-route messaging.
- The trace-route message requests both accumulation and an activity report, and the queue manager is not enabled for activity recording and trace route messaging.
- The trace-route message requests neither accumulation nor an activity report.

For every unrecorded activity the parameter, *UnrecordedActivities*, increments by one.

DiscontinuityCount
Specifies the number of times the trace-route message has been routed through a queue manager with applications that were not enabled for trace-route messaging. This value is incremented by the queue manager. If this value is greater than 0, only a partial message route can be determined.

MaxActivities
Specifies the maximum number of activities that can be performed on behalf of the trace-route message.

The total number of activities is the sum of *RecordedActivities*, *UnrecordedActivities*, and *DiscontinuityCount*. The total number of activities must not exceed the value of *MaxActivities*.

The value of *MaxActivities* can be:

A positive integer
The maximum number of activities.

If the maximum number of activities is exceeded, the trace-route message is rejected with feedback MQFB_MAX_ACTIVITIES. This can prevent the trace-route message from being forwarded indefinitely if caught in an infinite loop.

MQROUTE_UNLIMITED_ACTIVITIES
An unlimited number of activities can be performed on behalf of the trace-route message.

Accumulate
Specifies the method used to accumulate activity information. The value can be:

MQROUTE_ACCUMULATE_IN_MSG
If the queue manager is enabled for trace-route messaging, activity information is accumulated in the message data of the trace-route message.

If this value is specified, the trace-route message data consists of the following:

- The *TraceRoute* PCF group.
- Zero or more *Activity* PCF groups.

MQROUTE_ACCUMULATE_AND_REPLY
If the queue manager is enabled for trace-route messaging, activity information is accumulated in the message data of the trace-route message, and a trace-route reply message is generated if any of the following occur:

- The trace-route message is discarded by a WebSphere MQ Version 6 (or later) queue manager.
- The trace-route message is put to a local queue (target queue or dead-letter queue) by a WebSphere MQ Version 6 (or later) queue manager.

- The number of activities performed on the trace-route message exceeds the value of *MaxActivities*.

If this value is specified, the trace-route message data consists of the following:

- The *TraceRoute* PCF group.
- Zero or more *Activity* PCF groups.

MQROUTE_ACCUMULATE_NONE
Activity information is not accumulated in the message data of the trace-route message.

If this value is specified, the trace-route message data consists of the following:

- The *TraceRoute* PCF group.

Forward
Specifies where a trace-route message can be forwarded to. The value can be:

MQROUTE_FORWARD_IF_SUPPORTED
The trace-route message is only forwarded to queue managers that will honor the value of the *Deliver* parameter from the *TraceRoute* group.

MQROUTE_FORWARD_ALL
The trace-route message is forwarded to any queue manager, regardless of whether the value of the *Deliver* parameter will be honored.

Queue managers use the following algorithm when determining whether to forward a trace-route message to a remote queue manager:

1. Determine whether the remote queue manager is capable of supporting trace-route messaging.

   - If the remote queue manager is capable of supporting trace-route messaging, the algorithm continues to step .
   - If the remote queue manager is not capable of supporting trace-route messaging, the algorithm continues to step

2. Determine whether the *Deliver* parameter from the *TraceRoute* group contains any unrecognized delivery options in the MQROUTE_DELIVER_REJ_UNSUP_MASK bit mask.

   - If any unrecognized delivery options are found, the trace-route message is rejected with feedback MQFB_UNSUPPORTED_DELIVERY.
   - If no unrecognized delivery options are found, the algorithm continues to step .

3. Determine the value of the parameter *Deliver* from the *TraceRoute* PCF group in the trace-route message.

   - If *Deliver* is specified as MQROUTE_DELIVER_YES, the trace-route message is forwarded to the remote queue manager.
   - If *Deliver* is specified as MQROUTE_DELIVER_NO, the algorithm continues to step .

4. Determine whether the *Forward* parameter from the *TraceRoute* group contains any unrecognized forwarding options in the MQROUTE_FORWARDING_REJ_UNSUP_MASK bit mask.

   - If any unrecognized forwarding options are found, the trace-route message is rejected with feedback MQFB_UNSUPPORTED_FORWARDING.
   - If no unrecognized forwarding options are found, the algorithm continues to step .

5. Determine the value of the parameter *Forward* from the *TraceRoute* PCF group in the trace-route message.

   - If *Forward* is specified as MQROUTE_FORWARD_IF_SUPPORTED, the trace-route message is rejected with feedback MQFB_NOT_FORWARDED.
   - If *Forward* is specified as MQROUTE_FORWARD_ALL, trace-route message can be forwarded to the remote queue manager.

Deliver
> Specifies the action to be taken if the trace-route message reaches its intended destination. User-written applications must check this attribute before placing a trace-route message on its target queue. The value can be:
>
> **MQROUTE_DELIVER_YES**
>> On arrival, the trace-route message is put on the target queue. Any application performing a get operation on the target queue can retrieve the trace-route message.
>
> **MQROUTE_DELIVER_NO**
>> On arrival, the trace-route message is not delivered to the target queue. The message is processed according to its report options.

Setting up a common queue for trace-route reply messages

To determine the locations of the trace-route reply messages related to a specific message when the reports are delivered to the local system queue, it is more efficient to use a common queue on a single node

Before you begin

Set the ROUTEREC parameter to enable the queue manager for trace-route messaging and to specify that any trace-route reply messages generated are delivered to the local system queue SYSTEM.ADMIN.TRACE.ROUTE.QUEUE.

About this task

If a number of queue managers in a queue manager network are set to deliver trace-route reply messages to the local system queue, it can be time consuming to determine the locations of the trace-route reply messages related to a specific message. Alternatively, use a single node, which is a queue manager that hosts a common queue. All the queue managers in a queue manager network can deliver trace-route reply messages to this common queue. The benefit of using a common queue is that queue managers do not have to deliver trace-route reply messages to the reply-to queue specified in a message and, when determining the locations of the trace-route reply messages related to a message, you query one queue only.

To set up a common queue, perform the following steps:

Procedure

1. Select or define a queue manager as the single node
2. On the single node, select or define a queue for use as the common queue
3. On all queue managers that forward trace-route reply messages to the common queue, redefine the local system queue SYSTEM.ADMIN.TRACE.ROUTE.QUEUE as a remote queue definition

   a) Specify the name of the single node as the remote queue manager name

   b) Specify the name of the common queue as the remote queue name

Acquiring and using recorded information

Use any of the following techniques to acquire recorded activity information for a trace-route message

Note that the circumstances in which activity information is not acquired apply also to trace-route reply messages.

Activity information is not recorded when a trace-route message is processed by a queue manager that is disabled for both activity recording and trace-route messaging.

Acquiring information from trace-route reply messages

To acquire activity information you locate the trace-route reply message. Then you retrieve the message and analyze the activity information.

About this task

You can acquire activity information from a trace-route reply message only if you know the location of the trace-route reply message. Locate the message and process the activity information as follows:

Procedure

1. Check the reply-to queue that was specified in the message descriptor of the trace-route message. If the trace-route reply message is not on the reply-to queue, check the following locations:

   - The local system queue, SYSTEM.ADMIN.TRACE.ROUTE.QUEUE, on the target queue manager of the trace-route message

   - The common queue, if you have set up a common queue for trace-route reply messages

   - The local system queue, SYSTEM.ADMIN.TRACE.ROUTE.QUEUE, on any other queue manager in the queue manager network, which can occur if the trace-route message has been put to a dead-letter queue, or the maximum number of activities was exceeded

2. Retrieve the trace-route reply message

3. Use the WebSphere MQ display route application to display the recorded activity information

4. Study the activity information and obtain the information that you need

Acquiring information from trace-route messages

To acquire activity information you locate the trace-route message, which must have the appropriate parameters in the *TraceRoute* PCF group. Then you retrieve the message and analyze the activity information.

About this task

You can acquire activity information from a trace-route message only if you know the location of the trace-route message and it has the parameter *Accumulate* in the *TraceRoute* PCF group specified as either MQROUTE_ACCUMULATE_IN_MSG or MQROUTE_ACCUMULATE_AND_REPLY.

For the trace-route message to be delivered to the target queue the *Deliver* parameter in the *TraceRoute* PCF group must be specified as MQROUTE_DELIVER_YES.

Procedure

1. Check the target queue. If the trace-route message is not on the target queue, you can try to locate the trace-route message using a trace-route message enabled for activity recording. With the generated activity reports try to determine the last known location of the trace-route message.

2. Retrieve the trace-route message

3. Use the WebSphere MQ display route application to display the recorded activity information

4. Study the activity information and obtain the information that you need

Acquiring information from activity reports

To acquire activity information you locate the activity report, which must have the report option specified in the message descriptor. Then you retrieve the activity report and analyze the activity information.

About this task

You can acquire activity information from an activity report only if you know the location of the activity report and the report option MQRO_ACTIVITY was specified in the message descriptor of the trace-route message.

Procedure

1. Locate and order the activity reports generated for a trace-route message.

   When you have located the activity reports, you can order them manually or use the WebSphere MQ display route application to order and display the activity information automatically.

2. Study the activity information and obtain the information that you need

Additional activity information

As a trace-route message is routed through a queue manager network, user applications can record additional information by including one or more additional PCF parameters when writing the *Activity* group to the message data of the trace-route message or activity report.

Additional activity information can help system administrators to identify the route taken by a trace-route message took, or why that route was taken.

If you use the IBM WebSphere MQ display route application to display the recorded information for a trace-route message, any additional PCF parameters can only be displayed with a numeric identifier, unless the parameter identifier of each parameter is recognized by the IBM WebSphere MQ display route application. To recognize a parameter identifier, additional information must be recorded using the following PCF parameters. Include these PCF parameters in an appropriate place in the *Activity* PCF group.

GroupName

Table 13. Group name

Description	**Grouped parameters specifying the additional information.**
Identifier	MQGACF_VALUE_NAMING.
Data type	MQCFGR
Parameters in group	*ParameterName* *ParameterValue*

ParameterName

Table 14. Parameter name

Description	**Contains the name to be displayed by the IBM WebSphere MQ display route application, which puts the value of *ParameterValue* into context.**
Identifier	MQCA_VALUE_NAME.
Data type	MQCFST
Included in PCF group:	*GroupName*.
Value:	The name to be displayed.

ParameterValue

Table 15. Parameter value

Description	**Contains the value to be displayed by the IBM WebSphere MQ display route application.**
Identifier:	The PCF structure identifier for the additional information.
Data type:	The PCF structure data type for the additional information.

Table 15. Parameter value (continued)	
Description	**Contains the value to be displayed by the IBM WebSphere MQ display route application.**
Included in PCF group:	*GroupName.*
Value:	The value to be displayed.

Examples of recording additional activity information

The following examples illustrate how a user application can record additional information when performing an activity on behalf of a trace-route message. In both examples, the IBM WebSphere MQ display route application is used to generate a trace-route message, and display the activity information returned to it.

Example 1

Additional activity information is recorded by a user application in a format where the parameter identifier *is not* recognized by the WebSphere MQ display route application.

1. The WebSphere MQ display route application is used to generate and put a trace-route message into a queue manager network. The necessary options are set to request the following:

   - Activity information is accumulated in the message data of the trace-route message.
   - On arrival at the target queue the trace-route message is discarded, and a trace-route reply message is generated and delivered to a specified reply-to queue.
   - On receipt of the trace-route reply message, the WebSphere MQ display route application displays the accumulated activity information.

   The trace-route message is put into the queue manager network.

2. As the trace-route message is routed through the queue manager network a user application, that is enabled for trace-route messaging, performs a low detail activity on behalf of the message. In addition to writing the standard activity information to the trace-route message, the user application writes the following PCF parameter to the end of the Activity group:

   ColorValue

   > **Identifier**
   >> 65536

   > **Data type**
   >> MQCFST

   > **Value**
   >> 'Red'

   This additional PCF parameter gives further information about the activity that was performed, however it is written in a format where the parameter identifier *is not* recognized by the WebSphere MQ display route application.

3. The trace-route messages reaches the target queue and a trace-route reply message is returned to the WebSphere MQ display route application. The additional activity information is displayed as follows:

   ```
   65536: 'Red'
   ```

   The WebSphere MQ display route application does not recognize the parameter identifier of the PCF parameter and displays it as a numeric value. The context of the additional information is not clear.

   For an example of when the WebSphere MQ display route application does recognize the parameter identifier of the PCF parameter, see "Example 2" on page 75.

Example 2

Additional activity information is recorded by a user application in a format where the parameter identifier *is* recognized by the IBM WebSphere MQ display route application.

1. TheIBM WebSphere MQ display route application is used to generate and put a trace-route message into a queue manager network in the same fashion as in "Example 1" on page 74.

2. As the trace-route message is routed through the queue manager network a user application, that is enabled for trace-route messaging, performs a low detail activity on behalf of the message. In addition to writing the standard activity information to the trace-route message, the user application writes the following PCF parameters to the end of the Activity group:

ColorInfo

Table 16. Color information	
Description	**Grouped parameters specifying information about a color.**
Identifier:	MQGACF_VALUE_NAMING.
Data type:	MQCFGR.
Parameters in group:	*ColorName* *ColorValue*

ColorName

Table 17. Color name	
Description	**Contains the name to be displayed by the IBM WebSphere MQ display route application which puts the value of *ColorValue* into context.**
Identifier:	MQCA_VALUE_NAME.
Data type:	MQCFST.
Included in PCF group:	*ColorInfo*.
Value:	'Color'

ColorValue

Table 18. Color value	
Description	**Contains the value to be displayed by the IBM WebSphere MQ display route application.**
Identifier:	65536.
Data type:	MQCFST.
Included in PCF group:	*ColorInfo*.
Value:	'Red'

These additional PCF parameters gives further information about the activity that was performed. These PCF parameters are written in a format where the parameter identifier *is* recognized by the IBM WebSphere MQ display route application.

3. The trace-route messages reaches the target queue and a trace-route reply message is returned to the IBM WebSphere MQ display route application. The additional activity information is displayed as follows:

```
Color: 'Red'
```

The IBM WebSphere MQ display route application recognizes that the parameter identifier of the PCF structure containing the value of the additional activity information has a corresponding name. The corresponding name is displayed instead of the numeric value.

WebSphere MQ display route application

Use the WebSphere MQ display route application (**dspmqrte**) to work with trace-route messages and activity information related to a trace-route message, using a command-line interface.

Note: To run a Client Application against a queue manager, the Client Attachment feature must be installed.

You can use the WebSphere MQ display route application for the following purposes:

- To configure, generate, and put a trace-route message into a queue manager network.

  By putting a trace-route message into a queue manager network, activity information can be collected and used to determine the route that the trace-route message took. You can specify the characteristics of the trace-route messages as follows:

  – The destination of the trace-route message.

  – How the trace-route message mimics another message.

  – How the trace-route message should be handled as it is routed through a queue manager network.

  – Whether activity recording or trace-route messaging are used to record activity information.

- To order and display activity information related to a trace-route message.

  If the WebSphere MQ display route application has put a trace-route message into a queue manager network, after the related activity information has been returned, the information can be ordered and displayed immediately. Alternatively, the WebSphere MQ display route application can be used to order, and display, activity information related to a trace-route message that was previously generated.

Related reference
dspmqrte

Parameters for trace-route messages

Use this page to obtain an overview of the parameters provided by the WebSphere MQ display route application, **dspmqrte**, to determine the characteristics of a trace-route message, including how it is treated as it is routed through a queue manager network.
Related reference
dspmqrte

Queue manager connection
Use this page to specify the queue manager that the WebSphere MQ display route application connects to

-c
  Specifies that the WebSphere MQ display route application connects as a client application.

  If you do not specify this parameter, the WebSphere MQ display route application does not connect as a client application.

-m *QMgrName*
  The name of the queue manager to which the WebSphere MQ display route application connects. The name can contain up to 48 characters.

  If you do not specify this parameter, the default queue manager is used.

The target destination

Use this page to specify the target destination of a trace-route message

-q *TargetQName*
  If the WebSphere MQ display route application is being used to send a trace-route message into a
  queue manager network, *TargetQName* specifies the name of the target queue.

-ts *TargetTopicString*
  Specifies the topic string.

-qm *TargetQMgr*
  Qualifies the target destination; normal queue manager name resolution will then apply. The target
  destination is specified with *-q TargetQName* or *-ts TargetTopicString*.

  If you do not specify this parameter, the queue manager to which the WebSphere MQ display route
  application is connected is used as the target queue manager.

-o

  Specifies that the target destination is not bound to a specific destination. Typically this parameter is
  used when the trace-route message is to be put across a cluster. The target destination is opened with
  option MQOO_BIND_NOT_FIXED.

  If you do not specify this parameter, the target destination is bound to a specific destination.

The publication topic

For publish/subscribe applications, use this page to specify the topic string of a trace-route message for
the WebSphere MQ display route application to publish

-ts *TopicName*
  Specifies a topic string to which the WebSphere MQ display route application is to publish a trace-
  route message, and puts this application into topic mode. In this mode, the application traces all of
  the messages that result from the publish request.

You can also use the WebSphere MQ display route application to display the results from an activity report
that was generated for publish messages.

Message mimicking

Use this page to configure a trace-route message to mimic a message, for example when the original
message did not reach its intended destination

One use of trace-route messaging is to help determine the last known location of a message that did not
reach its intended destination. The IBM WebSphere MQ display route application provides parameters
that can help configure a trace-route message to mimic the original message. When mimicking a
message, you can use the following parameters:

-l *Persistence*
  Specifies the persistence of the generated trace-route message. Possible values for *Persistence* are:

  yes
      The generated trace-route message is persistent. (MQPER_PERSISTENT).

  no
      The generated trace-route message is **not** persistent. (MQPER_NOT_PERSISTENT).

  q
      The generated trace-route message inherits its persistence value from the destination specified
      by *-q TargetQName* or *-ts TargetTopicString*. (MQPER_PERSISTENCE_AS_Q_DEF).

A trace-route reply message, or any report messages, returned will share the same persistence value
as the original trace-route message.

If *Persistence* is specified as **yes**, you must specify the parameter *-rq ReplyToQ*. The reply-to queue
must not resolve to a temporary dynamic queue.

If you do not specify this parameter, the generated trace-route message is **not** persistent.

-p *Priority*

Specifies the priority of the trace-route message. The value of *Priority* is either greater than or equal to 0, or MQPRI_PRIORITY_AS_Q_DEF. MQPRI_PRIORITY_AS_Q_DEF specifies that the priority value is taken from the destination specified by *-q TargetQName* or *-ts TargetTopicString*.

If you do not specify this parameter, the priority value is taken from the destination specified by *-q TargetQName* or *-ts TargetTopicString*.

-xs *Expiry*

Specifies the expiry time for the trace-route message, in seconds.

If you do not specify this parameter, the expiry time is specified as 60 seconds.

-ro none | *ReportOption*

none

Specifies no report options are set.

ReportOption

Specifies report options for the trace-route message. Multiple report options can be specified using a comma as a separator. Possible values for *ReportOption* are:

activity

The report option MQRO_ACTIVITY is set.

coa

The report option MQRO_COA_WITH_FULL_DATA is set.

cod

The report option MQRO_COD_WITH_FULL_DATA is set.

exception

The report option MQRO_EXCEPTION_WITH_FULL_DATA is set.

expiration

The report option MQRO_EXPIRATION_WITH_FULL_DATA is set.

discard

The report option MQRO_DISCARD_MSG is set.

If neither *-ro ReportOption* nor *-ro none* are specified, then the MQRO_ACTIVITY and MQRO_DISCARD_MSG report options are specified.

The IBM WebSphere MQ display route application does not allow you to add user data to the trace-route message. If you require user data to be added to the trace-route message you must generate the trace-route message manually.

Recorded activity information

Use this page to specify the method used to return recorded activity information, which you can then use to determine the route that a trace-route message has taken

Recorded activity information can be returned as follows:

• In activity reports

• In a trace-route reply message

• In the trace-route message itself (having been put on the target queue)

When using **dspmqrte**, the method used to return recorded activity information is determined using the following parameters:

The `activity` report option, specified using `-ro`

Specifies that activity information is returned using activity reports. By default activity recording is enabled.

-ac -ar

Specifies that activity information is accumulated in the trace-route message, and that a trace-route reply message is to be generated.

-ac

Specifies that activity information is to be accumulated within the trace-route message.

If you do not specify this parameter, activity information is **not** accumulated within the trace-route message.

-ar

Requests that a trace-route reply message containing all accumulated activity information is generated in the following circumstances:

- The trace-route message is discarded by a IBM WebSphere MQ queue manager.
- The trace-route message is put to a local queue (target queue or dead-letter queue) by a IBM WebSphere MQ queue manager.
- The number of activities performed on the trace-route message exceeds the value of specified in *-s Activities*.

-ac -d yes

Specifies that activity information is accumulated in the trace-route message, and that on arrival, the trace-route message will be put on the target queue.

-ac

Specifies that activity information is to be accumulated within the trace-route message.

If you do not specify this parameter, activity information is **not** accumulated within the trace-route message.

-d yes

On arrival, the trace-route message is put to the target queue, even if the queue manager does not support trace-route messaging.

If you do not specify this parameter, the trace-route message is **not** put to the target queue.

The trace-route message can then be retrieved from the target queue, and the recorded activity information acquired.

You can combine these methods as required.

Additionally, the detail level of the recorded activity information can be specified using the following parameter:

-t *Detail*

Specifies the activities that are recorded. The possible values for *Detail* are:

low

Activities performed by user-defined application are recorded only.

medium

Activities specified in **low** are recorded. Additionally, publish activities and activities performed by MCAs are recorded.

high

Activities specified in **low**, and **medium** are recorded. MCAs do not expose any further activity information at this level of detail. This option is available to user-defined applications that are to expose further activity information only. For example, if a user-defined application determines the route a message takes by considering certain message characteristics, the routing logic could be included with this level of detail.

If you do not specify this parameter, medium level activities are recorded.

By default the IBM WebSphere MQ display route application uses a temporary dynamic queue to store the returned messages. When the IBM WebSphere MQ display route application ends, the temporary dynamic queue is closed, and any messages are purged. If the returned messages are required beyond the current

execution of the IBM WebSphere MQ display route application ends, then a permanent queue must be specified using the following parameters:

-rq *ReplyToQ*
    Specifies the name of the reply-to queue that all responses to the trace-route message are sent to. If the trace-route message is persistent, or if the *-n* parameter is specified, a reply-to queue must be specified that is **not** a temporary dynamic queue.

    If you do not specify this parameter then a dynamic reply-to queue is created using the system default model queue, SYSTEM.DEFAULT.MODEL.QUEUE.

-rqm *ReplyToQMgr*
    Specifies the name of the queue manager where the reply-to queue resides. The name can contain up to 48 characters.

    If you do not specify this parameter, the queue manager to which the IBM WebSphere MQ display route application is connected is used as the reply-to queue manager.

How the trace-route message is handled

Use this page to control how a trace-route message is handled as it is routed through a queue manager network.

The following parameters can restrict where the trace-route message can be routed in the queue manager network:

-d *Deliver*
    Specifies whether the trace-route message is to be delivered to the target queue on arrival. Possible values for *Deliver* are:

yes	On arrival, the trace-route message is put to the target queue, even if the queue manager does not support trace-route messaging.
no	On arrival, the trace-route message is **not** put to the target queue.

    If you do not specify this parameter, the trace-route message is **not** put to the target queue.

-f *Forward*
    Specifies the type of queue manager that the trace-route message can be forwarded to. For details of the algorithm that queue managers use to determine whether to forward a message to a remote queue manager, refer to"The TraceRoute PCF group" on page 68. The possible values for *Forward* are:

    all
        The trace-route message is forwarded to any queue manager.

        Warning: If forwarded to a IBM WebSphere MQ queue manager earlier than Version 6.0, the trace-route message will not be recognized and can be delivered to a local queue despite the value of the *-d Deliver* parameter.

    supported
        The trace-route message is only forwarded to a queue manager that will honor the *Deliver* parameter from the *TraceRoute* PCF group

    If you do not specify this parameter, the trace-route message will only be forwarded to a queue manager that will honor the *Deliver* parameter.

The following parameters can prevent a trace-route message from remaining in a queue manager network indefinitely:

-s *Activities*
    Specifies the maximum number of recorded activities that can be performed on behalf of the trace-route message before it is discarded. This prevents the trace-route message from being forwarded indefinitely if caught in an infinite loop. The value of *Activities* is either greater than or equal to 1, or MQROUTE_UNLIMITED_ACTIVITIES. MQROUTE_UNLIMITED_ACTIVITIES specifies that an unlimited number of activities can be performed on behalf of the trace-route message.

If you do not specify this parameter, an unlimited number of activities can be performed on behalf of the trace-route message.

-xs *Expiry*
Specifies the expiry time for the trace-route message, in seconds.

If you do not specify this parameter, the expiry time is specified as 60 seconds.

-xp *PassExpiry*
Specifies whether the expiry time from the trace-route message is passed on to a trace-route reply message. Possible values for *PassExpiry* are:

yes
The report option MQRO_PASS_DISCARD_AND_EXPIRY is specified in the message descriptor of the trace-route message.

If a trace-route reply message, or activity reports, are generated for the trace-route message, the MQRO_DISCARD report option (if specified), and the remaining expiry time are passed on.

This is the default value.

no
The report option MQRO_PASS_DISCARD_AND_EXPIRY is not specified.

If a trace-route reply message is generated for the trace-route message, the discard option and expiry time from the trace-route message are **not** passed on.

If you do not specify this parameter, MQRO_PASS_DISCARD_AND_EXPIRY is not specified.

The `discard` report option, specified using `-ro`
Specifies the MQRO_DISCARD_MSG report option. This can prevent the trace-route message remaining in the queue manager network indefinitely.

Display of activity information

The IBM WebSphere MQ display route application can display activity information for a trace-route message that it has just put into a queue manager network, or it can display activity information for a previously generated trace-route message. It can also display additional information recorded by user-written applications.

To specify whether activity information returned for a trace-route message is displayed, specify the following parameter:

-n
Specifies that activity information returned for the trace-route message is not to be displayed.

If this parameter is accompanied by a request for a trace-route reply message, (*-ar*), or any of the report generating options from (*-ro ReportOption*), then a specific (non-model) reply-to queue must be specified using *-rq ReplyToQ*. By default, only activity report messages are requested.

After the trace-route message is put to the specified target queue, a 48 character hexadecimal string is displayed containing the message identifier of the trace-route message. The message identifier can be used by the IBM WebSphere MQ display route application to display the activity information for the trace-route message at a later time, using the *-i CorrelId* parameter.

If you do not specify this parameter, activity information returned for the trace-route message is displayed in the form specified by the *-v* parameter.

When displaying activity information for a trace-route message that has just been put into a queue manager network, the following parameter can be specified:

-w *WaitTime*
Specifies the time, in seconds, that the IBM WebSphere MQ display route application will wait for activity reports, or a trace-route reply message, to return to the specified reply-to queue.

If you do not specify this parameter, the wait time is specified as the expiry time of the trace-route message, plus 60 seconds.

When displaying previously accumulated activity information the following parameters must be set:

-q *TargetQName*
> If the IBM WebSphere MQ display route application is being used to view previously gathered activity information, *TargetQName* specifies the name of the queue where the activity information is stored.

-i `CorrelId`
> This parameter is used when the IBM WebSphere MQ display route application is used to display previously accumulated activity information only. There can be many activity reports and trace-route reply messages on the queue specified by *-q TargetQName*. *CorrelId* is used to identify the activity reports, or a trace-route reply message, related to a trace-route message. Specify the message identifier of the original trace-route message in *CorrelId*.
>
> The format of *CorrelId* is a 48 character hexadecimal string.

The following parameters can be used when displaying previously accumulated activity information, or when displaying current activity information for a trace-route message:

-b
> Specifies that the IBM WebSphere MQ display route application will only browse activity reports or a trace-route reply message related to a message. This allows activity information to be displayed again at a later time.
>
> If you do not specify this parameter, the IBM WebSphere MQ display route application will destructively get activity reports or a trace-route reply message related to a message.

-v summary | all | none | outline `DisplayOption`

summary
> The queues that the trace-route message was routed through are displayed.

all
> All available information is displayed.

none
> No information is displayed.

outline *DisplayOption*
> Specifies display options for the trace-route message. Multiple display options can be specified using a comma as a separator.
>
> If no values are supplied the following is displayed:
>
> - The application name
> - The type of each operation
> - Any operation specific parameters
>
> Possible values for *DisplayOption* are:
>
> **activity**
> > All non-PCF group parameters in *Activity* PCF groups are displayed.
>
> **identifiers**
> > Values with parameter identifiers MQBACF_MSG_ID or MQBACF_CORREL_ID are displayed. This overrides *msgdelta*.
>
> **message**
> > All non-PCF group parameters in *Message* PCF groups are displayed. When this value is specified, you cannot specify *msgdelta*.
>
> **msgdelta**
> > All non-PCF group parameters in *Message* PCF groups, that have changed since the last operation, are displayed. When this value is specified, you cannot specify *message*.
>
> **operation**
> > All non-PCF group parameters in *Operation* PCF groups are displayed.
>
> **traceroute**
> > All non-PCF group parameters in *TraceRoute* PCF groups are displayed.

If you do not specify this parameter, a summary of the message route is displayed.

Display of additional information

As a trace-route message is routed through a queue manager network, user-written applications can record additional information by writing one or more additional PCF parameters to the message data of the trace-route message or to the message data of an activity report. For the IBM WebSphere MQ display route application to display additional information in a readable form it must be recorded in a specific format, as described in "Additional activity information" on page 73.

WebSphere MQ display route application examples

The following examples show how you can use the WebSphere MQ display route application. In each example, two queue managers (QM1 and QM2) are inter-connected by two channels (QM2.TO.QM1 and QM1.TO.QM2).

Example 1 - Requesting activity reports

Display activity information from a trace-route message delivered to the target queue

In this example the WebSphere MQ display route application connects to queue manager, QM1, and is used to generate and deliver a trace-route message to the target queue, TARGET.Q, on remote queue manager, QM2. The necessary report option is specified so that activity reports are requested as the trace-route reply message is routed. On arrival at the target queue the trace-route message is discarded. Activity information returned to the WebSphere MQ display route application using activity reports is put in order and displayed.
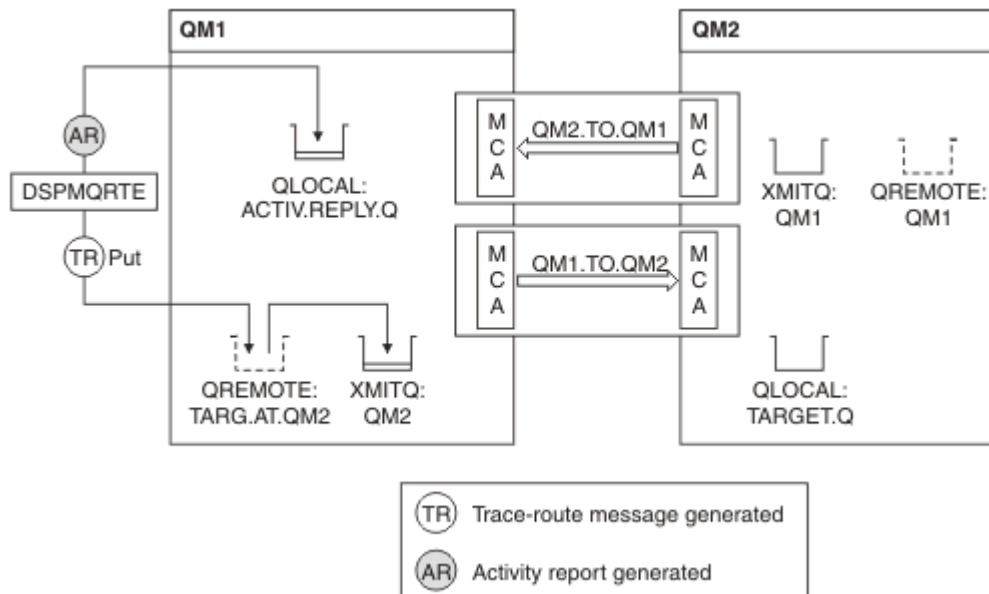
Figure 9. Requesting activity reports, Diagram 1

- The ACTIVREC attribute of each queue manager (QM1 and QM2) is set to MSG.
- The following command is issued:

```
dspmqrte -m QM1 -q TARG.AT.QM2 -rq ACTIV.REPLY.Q
```

QM1 is the name of the queue manager to which the WebSphere MQ display route application connects, TARG.AT.QM2 is the name of the target queue, and ACTIV.REPLY.Q is the name of the queue to which it is requested that all responses to the trace-route message are sent.

Default values are assumed for all options that are not specified, but note in particular the -f option (the trace-route message is forwarded only to a queue manager that honors the Deliver parameter of

the TraceRoute PCF group), the -d option (on arrival, the trace-route message is not put on the target queue), the -ro option (MQRO_ACTIVITY and MQRO_DISCARD_MSG report options are specified), and the -t option (medium detail level activity is recorded).

- DSPMQRTE generates the trace-route message and puts it on the remote queue TARG.AT.QM2.
- DSPMQRTE then looks at the value of the ACTIVREC attribute of queue manager QM1. The value is MSG, therefore DSPMQRTE generates an activity report and puts it on the reply queue ACTIV.REPLY.Q.
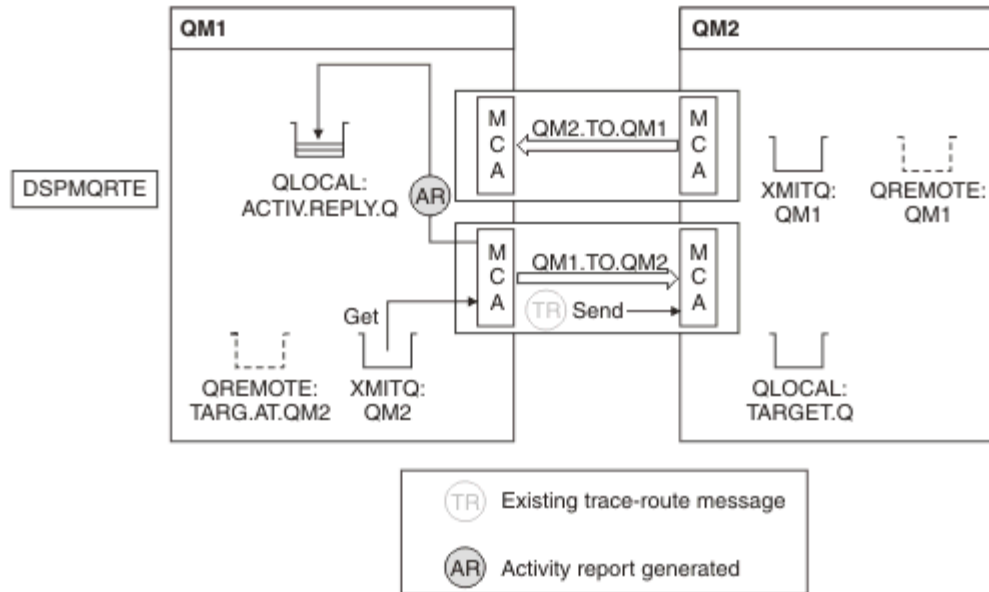
Figure 10. Requesting activity reports, Diagram 2

- The sending message channel agent (MCA) gets the trace-route message from the transmission queue. The message is a trace-route message, therefore the MCA begins to record the activity information.
- The ACTIVREC attribute of the queue manager (QM1) is MSG, and the MQRO_ACTIVITY option is specified in the Report field of the message descriptor, therefore the MCA will later generate an activity report. The RecordedActivities parameter value in the TraceRoute PCF group is incremented by 1.
- The MCA checks that the MaxActivities value in the TraceRoute PCF group has not been exceeded.
- Before the message is forwarded to QM2 the MCA follows the algorithm that is described in Forwarding (steps "1" on page 70, "4" on page 70, and "5" on page 70) and the MCA chooses to send the message.
- The MCA then generates an activity report and puts it on the reply queue (ACTIV.REPLY.Q).
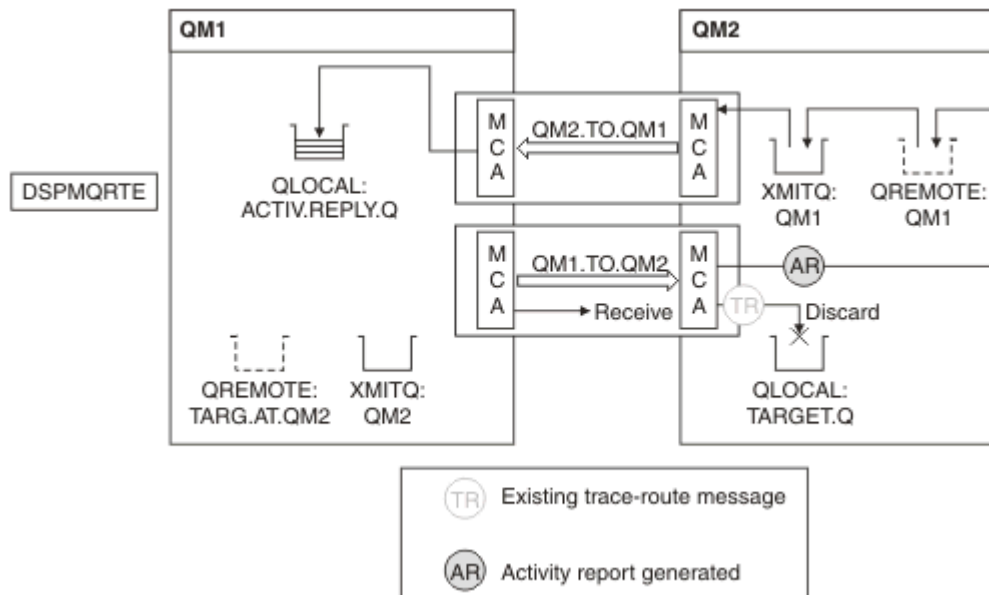
Figure 11. Requesting activity reports, Diagram 3

- The receiving MCA receives the trace-route message from the channel. The message is a trace-route message, therefore the MCA begins to record the information about the activity.
- If the queue manager that the trace-route message has come from is Version 5.3.1 or earlier, the MCA increments the DiscontinuityCount parameter of the TraceRoute PCF by 1. This is not the case here.
- The ACTIVREC attribute of the queue manager (QM2) is MSG, and the MQRO_ACTIVITY option is specified, therefore the MCA will generate an activity report. The RecordedActivities parameter value is incremented by 1.
- The target queue is a local queue, therefore the message is discarded with feedback MQFB_NOT_DELIVERED, in accordance with the Deliver parameter value in the TraceRoute PCF group.
- The MCA then generates the final activity report and puts it on the reply queue. This resolves to the transmission queue that is associated with queue manager QM1 and the activity report is returned to queue manager QM1 (ACTIV.REPLY.Q).
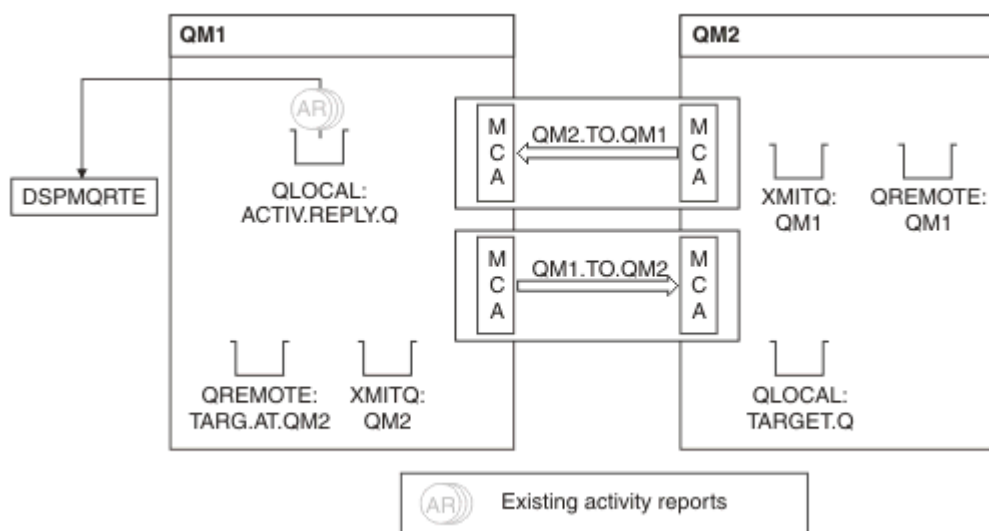
Figure 12. Requesting activity reports, Diagram 4

- Meanwhile, DSPMQRTE has been continually performing MQGETs on the reply queue (ACTIV.REPLY.Q), waiting for activity reports. It will wait for up to 120 seconds (60 seconds longer than the expiry time of the trace-route message) since -w was not specified when DSPMQRTE was started.
- DSPMQRTE gets the 3 activity reports off the reply queue.
- The activity reports are ordered using the RecordedActivities, UnrecordedActivities, and DiscontinuityCount parameters in the TraceRoute PCF group for each of the activities. The only value that is non-zero in this example is RecordedActivities, therefore this is the only parameter that is actually used.
- The program ends as soon as the discard operation is displayed. Even though the final operation was a discard, it is treated as though a put took place because the feedback is MQFB_NOT_DELIVERED.

The output that is displayed follows:

```
AMQ8653: DSPMQRTE command started with options '-m QM1 -q TARG.AT.QM2
 -rq ACTIV.REPLY.Q'.
AMQ8659: DSPMQRTE command successfully put a message on queue 'QM2',
 queue manager 'QM1'.
AMQ8674: DSPMQRTE command is now waiting for information to display.
AMQ8666: Queue 'QM2' on queue manager 'QM1'.
AMQ8666: Queue 'TARGET.Q' on queue manager 'QM2'.
AMQ8652: DSPMQRTE command has finished.
```

Example 2 - Requesting a trace-route reply message
Generate and deliver a trace-route message to the target queue

In this example the WebSphere MQ display route application connects to queue manager, QM1, and is used to generate and deliver a trace-route message to the target queue, TARGET.Q, on remote queue manager, QM2. The necessary option is specified so that activity information is accumulated in the trace-route message. On arrival at the target queue a trace-route reply message is requested, and the trace-route message is discarded.
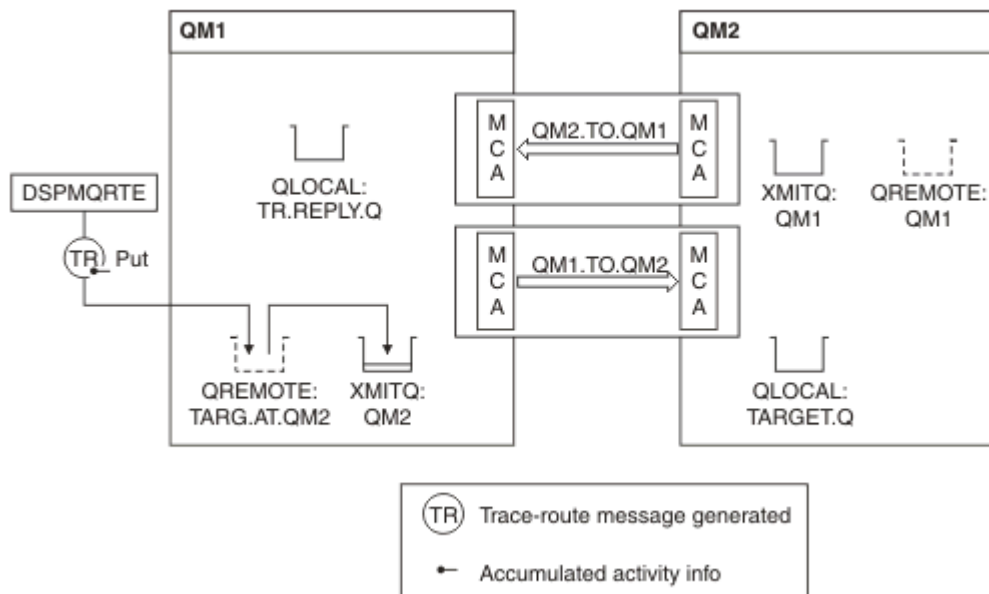
Figure 13. Requesting a trace-route reply message, Diagram 1

- The ROUTEREC attribute of each queue manager (QM1 and QM2) is set to MSG.
- The following command is issued:

```
dspmqrte -m QM1 -q TARG.AT.QM2 -rq TR.REPLY.Q -ac -ar -ro discard
```

QM1 is the name of the queue manager to which the WebSphere MQ display route application connects, TARG.AT.QM2 is the name of the target queue, and ACTIV.REPLY.Q is the name of the queue to which it is requested that all responses to the trace-route message are sent. The -ac option specifies that activity information is accumulated in the trace-route message, the -ar option specifies that all accumulated activity is sent to the reply-to queue that is specified by the -rq option (that is, TR.REPLY.Q). The -ro option specifies that report option MQRO_DISCARD_MSG is set which means that activity reports are not generated in this example.

- DSPMQRTE accumulates activity information in the trace-route message before the message is put on the target route. The queue manager attribute ROUTEREC must not be DISABLED for this to happen.

Figure 14. Requesting a trace-route reply message, Diagram 2

- The message is a trace-route message, therefore the sending MCA begins to record information about the activity.
- The queue manager attribute ROUTEREC on QM1 is not DISABLED, therefore the MCA accumulates the activity information within the message, before the message is forwarded to queue manager QM2.

Figure 15. Requesting a trace-route reply message, Diagram 3

- The message is a trace-route message, therefore the receiving MCA begins to record information about the activity.
- The queue manager attribute ROUTEREC on QM2 is not DISABLED, therefore the MCA accumulates the information within the message.
- The target queue is a local queue, therefore the message is discarded with feedback MQFB_NOT_DELIVERED, in accordance with the Deliver parameter value in the TraceRoute PCF group.
- This is the last activity that will take place on the message, and because the queue manager attribute ROUTEREC on QM1 is not DISABLED, the MCA generates a trace-route reply message in accordance with the Accumulate value. The value of ROUTEREC is MSG, therefore the reply message is put on the reply queue. The reply message contains all the accumulated activity information from the trace-route message.
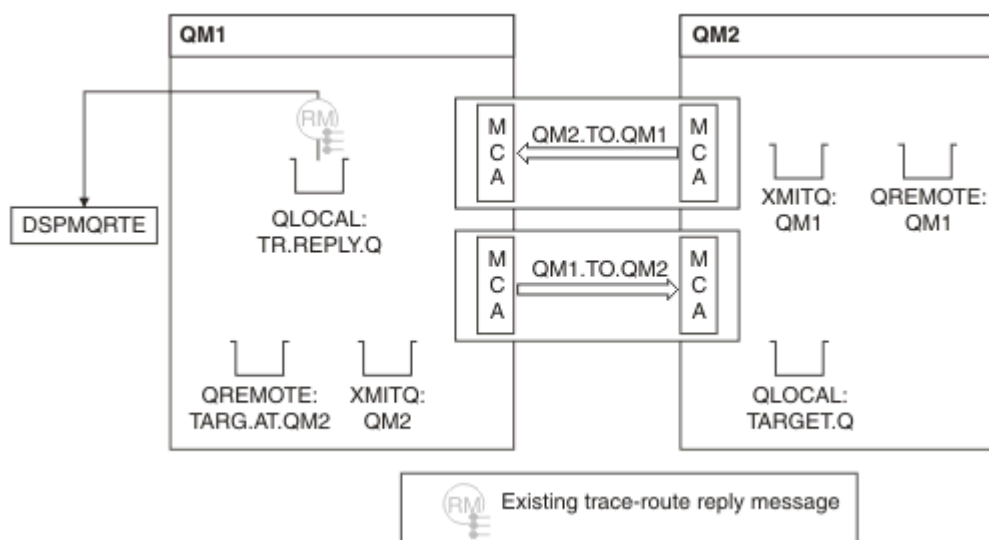
Figure 16. Requesting a trace-route reply message, Diagram 4

- Meanwhile DSPMQRTE is waiting for the trace-route reply message to return to the reply queue. When it returns, DSPMQRTE parses each activity that it contains and prints it out. The final operation is a discard operation. DSPMQRTE ends after it has been printed.

The output that is displayed follows:

```
AMQ8653: DSPMQRTE command started with options '-m QM1 -q TARG.AT.QM2 -rq
 TR.REPLY.Q'.
AMQ8659: DSPMQRTE command successfully put a message on queue 'QM2', queue
 manager 'QM1'.
AMQ8674: DSPMQRTE command is now waiting for information to display.
AMQ8666: Queue 'QM2' on queue manager 'QM1'.
AMQ8666: Queue 'TARGET.Q' on queue manager 'QM2'.
AMQ8652: DSPMQRTE command has finished.
```

Example 3 - Delivering activity reports to the system queue

Detect when activity reports are delivered to queues other than the reply-to queue and use the WebSphere MQ display route application to read activity reports from the other queue.

This example is the same as "Example 1 - Requesting activity reports" on page 83, except that QM2 now has the value of the ACTIVREC queue manage attribute set to QUEUE. Channel QM1.TO.QM2 must have been restarted for this to take effect.

This example demonstrates how to detect when activity reports are delivered to queues other than the reply-to queue. Once detected, the WebSphere MQ display route application is used to read activity reports from another queue.
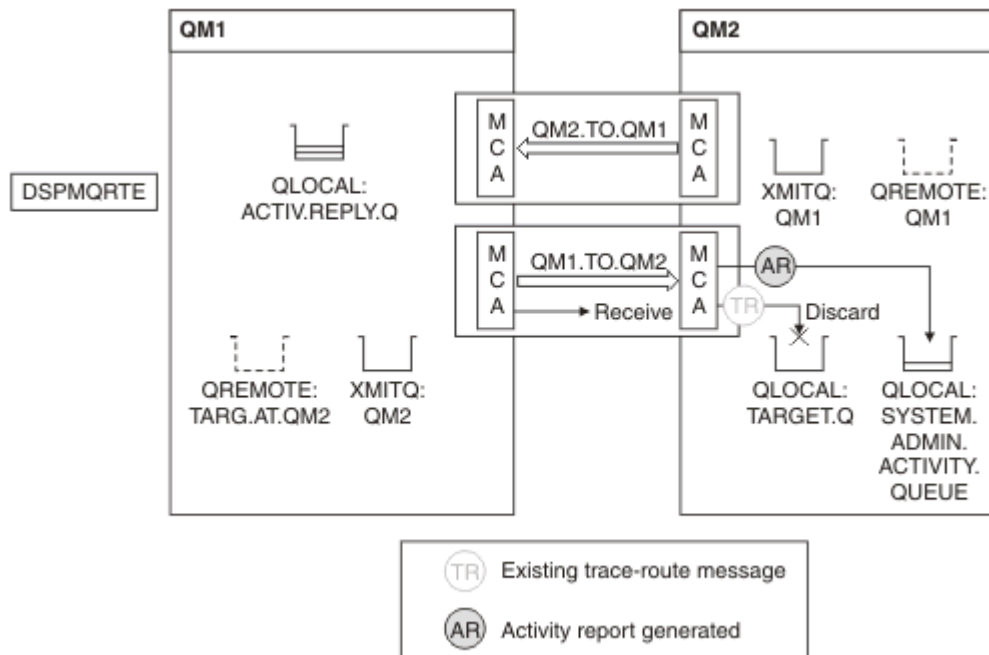
Figure 17. Delivering activity reports to the system queue, Diagram 1

- The message is a trace-route message, therefore the receiving MCA begins to record information about the activity.
- The value of the ACTIVREC queue manager attribute on QM2 is now QUEUE, therefore the MCA generates an activity report, but puts it on the system queue (SYSTEM.ADMIN.ACTIVITY.QUEUE) and not on the reply queue (ACTIV.REPLY.Q).
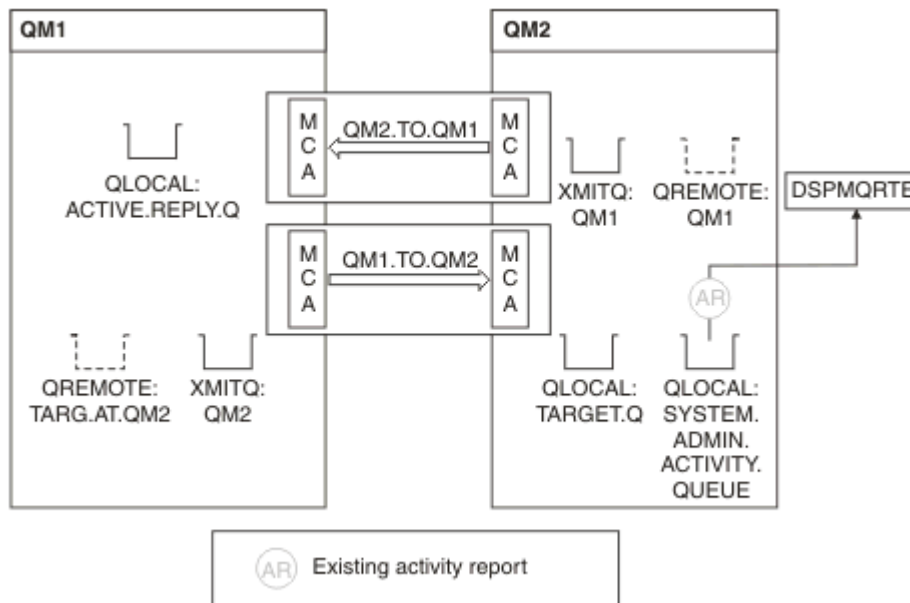
Figure 18. Delivering activity reports to the system queue, Diagram 2

- Meanwhile DSPMQRTE has been waiting for activity reports to arrive on ACTIV.REPLY.Q. Only two arrive. DSPMQRTE continues waiting for 120 seconds because it seems that the route is not yet complete.

The output that is displayed follows:

```
AMQ8653: DSPMQRTE command started with options '-m QM1 -q TARG.AT.QM2 -rq
         ACTIV.REPLY.Q -v outline identifiers'.
AMQ8659: DSPMQRTE command successfully put a message on queue 'QM2', queue
         manager 'QM1'.
AMQ8674: DSPMQRTE command is now waiting for information to display.
--------------------------------------------------------------------------
Activity:
 ApplName: 'cann\output\bin\dspmqrte.exe'

 Operation:
  OperationType: Put

  Message:

   MQMD:
    MsgId: X'414D51204C415247455512020202020202020A3C9154220001502'
    CorrelId: X'414D51204C4152474555120202020202020A3C9154220001503'
  QMgrName: 'QM1                                              '
  QName: 'TARG.AT.QM2                                    '
  ResolvedQName: 'QM2                                         '
  RemoteQName: 'TARGET.Q                                    '
  RemoteQMgrName: 'QM2                                         '
--------------------------------------------------------------------------
Activity:
 ApplName: 'cann\output\bin\runmqchl.EXE'

 Operation:
  OperationType: Get

  Message:

   MQMD:
    MsgId: X'414D51204C415247455512020202020202020A3C9154220001505'
    CorrelId: X'414D51204C415247455512020202020202020A3C9154220001502'

   EmbeddedMQMD:
    MsgId: X'414D51204C415247455512020202020202020A3C9154220001502'
    CorrelId: X'414D51204C415247455512020202020202020A3C9154220001503'
  QMgrName: 'QM1                                              '
  QName: 'QM2                                            '
  ResolvedQName: 'QM2                                  '

 Operation:
  OperationType: Send

  Message:

   MQMD:
    MsgId: X'414D51204C415247455512020202020202020A3C9154220001502'
    CorrelId: X'414D51204C415247455512020202020202020A3C9154220001503'
  QMgrName: 'QM1                                              '
  RemoteQMgrName: 'QM2                                       '
  ChannelName: 'QM1.TO.QM2          '
  ChannelType: Sender
  XmitQName: 'QM2                                        '
--------------------------------------------------------------------------
AMQ8652: DSPMQRTE command has finished.
```

- The last operation that DSPMQRTE observed was a Send, therefore the channel is running. Now we must work out why we did not receive any more activity reports from queue manager QM2 (as identified in RemoteQMgrName).
- To check whether there is any activity information on the system queue, start DSPMQRTE on QM2 to try and collect more activity reports. Use the following command to start DSPMQRTE:

```
dspmqrte -m QM2 -q SYSTEM.ADMIN.ACTIVITY.QUEUE
        -i 414D51204C415247455512020202020202020A3C9154220001502 -v outline
```

where 414D51204C415247455512020202020202020A3C9154220001502 is the MsgId of the trace-route message that was put.

- DSPMQRTE then performs a sequence of MQGETs again, waiting for responses on the system activity queue related to the trace-route message with the specified identifier.

- DSPMQRTE gets one more activity report, which it displays. DSPMQRTE determines that the preceding activity reports are missing, and displays a message saying this. We already know about this part of the route, however.

The output that is displayed follows:

```
AMQ8653: DSPMQRTE command started with options '-m QM2
         -q SYSTEM.ADMIN.ACTIVITY.QUEUE
         -i 414D51204C415247455120202020202020A3C915420001502 -v outline'.
AMQ8674: DSPMQRTE command is now waiting for information to display.
--------------------------------------------------------------------------------

Activity:
 Activity information unavailable.

--------------------------------------------------------------------------------
Activity:
 ApplName: 'cann\output\bin\AMQRMPPA.EXE'

 Operation:
  OperationType: Receive
  QMgrName: 'QM2                                             '
  RemoteQMgrName: 'QM1                                            '
  ChannelName: 'QM1.TO.QM2           '
  ChannelType: Receiver

 Operation:
  OperationType: Discard
  QMgrName: 'QM2                                             '
  QName: 'TARGET.Q                                       '
  Feedback: NotDelivered

--------------------------------------------------------------------------------
AMQ8652: DSPMQRTE command has finished.
```

- This activity report indicates that the route information is now complete. No problem occurred.
- Just because route information is unavailable, or because DSPMQRTE cannot display all of the route, this does not mean that the message was not delivered. For example, the queue manager attributes of different queue managers might be different, or a reply queue might not be defined to get the response back.

Example 4 - Diagnosing a channel problem
Diagnose a problem in which the trace-route message does not reach the target queue

In this example the WebSphere MQ display route application connects to queue manager, QM1, generates a trace-route message, then attempts to deliver it to the target queue, TARGET.Q, on remote queue manager, QM2. In this example the trace-route message does not reach the target queue. The available activity report is used to diagnose the problem.

Figure 19. Diagnosing a channel problem

- In this example, the channel QM1.TO.QM2 is not running.
- DSPMQRTE puts a trace-route message (as in example 1) to the target queue and generates an activity report.
- There is no MCA to get the message from the transmission queue (QM2), therefore this is the only activity report that DSPMQRTE gets back from the reply queue. This time the fact that the route is not complete does indicate a problem. The administrator can use the transmission queue found in ResolvedQName to investigate why the transmission queue is not being serviced.

The output that is displayed follows:

```
AMQ8653: DSPMQRTE command started with options '-m QM1 -q TARG.AT.QM2
         -rq ACTIV.REPLY.Q -v outline'.
AMQ8659: DSPMQRTE command successfully put a message on queue 'QM2',
         queue manager 'QM1'.
AMQ8674: DSPMQRTE command is now waiting for information to display.
--------------------------------------------------------------------------------
Activity:
 ApplName: 'cann\output\bin\dspmqrte.exe'

 Operation:
  OperationType: Put
  QMgrName: 'QM1                                             '
  QName: 'TARG.AT.QM2                                     '
  ResolvedQName: 'QM2                                     '
  RemoteQName: 'TARGET.Q                                  '
  RemoteQMgrName: 'QM2                                     '


--------------------------------------------------------------------------------
 AMQ8652: DSPMQRTE command has finished.
```

Activity report reference

Use this page to obtain an overview of the activity report message format. The activity report message data contains the parameters that describe the activity.

Activity report format

Activity reports are standard IBM WebSphere MQ report messages containing a message descriptor and message data. Activity reports are PCF messages generated by applications that have performed an activity on behalf of a message as it has been routed through a queue manager network.

Activity reports contain the following information:

A message descriptor
   An MQMD structure

Message data
   Consists of the following:

   - An embedded PCF header (MQEPH).

   - Activity report message data.

Activity report message data consists of the *Activity* PCF group and, if generated for a trace-route message, the *TraceRoute* PCF group.

shows the structure of these reports, including parameters that are returned only under certain conditions.

Table 19. Activity report format		
MQMD structure	**Embedded PCF header MQEPH structure**	**Activity report message data**

Structure identifier
Structure version
Report options
Message type
Expiration time
Feedback
Encoding
Coded character set ID
Message format
Priority
Persistence
Message identifier
Correlation identifier
Backout count
Reply-to queue
Reply-to queue manager
User identifier
Accounting token
Application identity data
Application type
Application name
Put date
Put time
Application origin data
Group identifier
Message sequence number
Offset
Message flags
Original length
	Structure identifier
	Structure version
	Structure length
	Encoding
	Coded character set ID
	Message format
	Flags
	PCF header (MQCFH)
	  Structure type
	  Structure length
	  Structure version
	  Command identifier
	  Message sequence number
	  Control options
	  Completion code
	  Reason code
	  Parameter count
		Activity
		  Activity application name
		  Activity application type
		  Activity description
		  Operation
		    Operation type
		    Operation date
		    Operation time
		    Message
		      Message length
		      MQMD [8]
		      EmbeddedMQMD
		    Queue manager name
		    Queue sharing group name
		    Queue name [1]
		[2] [3]
		[7]
		    Resolved queue name [1]
		[3]
		[7]
		    Remote queue name [3]
		[7]
		    Remote queue manager name [2]
		[3] [4]
		[5]
		[7]
		    Subscription level [9]
		    Subscription identifier
		[9]
		    Feedback [2]
		[10]
		    Channel name [4]
		[5]
		    Channel type [4]
		[5]
		    Transmission queue name [5]
		  TraceRoute [6]
		    Detail
		    Recorded activities
		    Unrecorded activities
		    Discontinuity count
		    Max activities
		    Accumulate
		    Deliver

Notes:

1. Returned for Get and Browse operations.

2. Returned for Discard operations.

3. Returned for Put, Put Reply, and Put Report operations.

4. Returned for Receive operations.

5. Returned for Send operations.

6. Returned for trace-route messages.

7. Not returned for Put operations to a topic, contained within Publish activities.

8. Not returned for Excluded Publish operations. For Publish and Discarded Publish operations, returned containing a subset of parameters.

9. Returned for Publish, Discarded Publish, and Excluded Publish operations.

10. Returned for Discarded Publish and Excluded Publish operations.

Activity report MQMD (message descriptor)

Use this page to view the values contained by the MQMD structure for an activity report

StrucId
Structure identifier:

Data type
MQCHAR4

Value
MQMD_STRUC_ID.

Version
Structure version number

Data type
MQLONG

Values
Copied from the original message descriptor. Possible values are:

MQMD_VERSION_1
Version-1 message descriptor structure, supported in all environments.

MQMD_VERSION_2
Version-2 message descriptor structure, supported on AIX®, HP-UX, z/OS, IBM i, Solaris, Linux, Windows, and all WebSphere MQ MQI clients connected to these systems.

Report
Options for further report messages

Data type
MQLONG

Value
If MQRO_PASS_DISCARD_AND_EXPIRY or MQRO_DISCARD_MSG were specified in the *Report* field of the original message descriptor:

MQRO_DISCARD
The report is discarded if it cannot be delivered to the destination queue.

Otherwise:

MQRO_NONE
No reports required.

MsgType
Indicates type of message

Data type
MQLONG

Value
MQMT_REPORT

Expiry
Report message lifetime

Data type
    MQLONG

Value
    If the *Report* field in the original message descriptor is specified as
    MQRO_PASS_DISCARD_AND_EXPIRY, the remaining expiry time from the original message is
    used.

    Otherwise:

MQEI_UNLIMITED
    The report does not have an expiry time.

Feedback

Description:	Feedback or reason code.
Data type:	MQLONG.
Value:	**MQFB_ACTIVITY** Activity report.

Encoding

Description:	Numeric encoding of report message data.
Data type:	MQLONG.
Value:	MQENC_NATIVE.

CodedCharSetId

Description:	Character set identifier of report message data.
Data type:	MQLONG.
Value:	Set as appropriate.

Format

Description:	Format name of report message data
Data type:	MQCHAR8.
Value:	**MQFMT_EMBEDDED_PCF** Embedded PCF message.

Priority

Description:	Report message priority.
Data type:	MQLONG.
Value:	Copied from the original message descriptor.

Persistence

Description:	Report message persistence.
Data type:	MQLONG.
Value:	Copied from the original message descriptor.

MsgId

Description:	Message identifier.

Data type:	MQBYTE24.
Values:	If the *Report* field in the original message descriptor is specified as MQRO_PASS_MSG_ID, the message identifier from the original message is used.
	Otherwise, a unique value will be generated by the queue manager.

CorrelId

Description:	Correlation identifier.
Data type:	MQBYTE24.
Value:	If the *Report* field in the original message descriptor is specified as MQRO_PASS_CORREL_ID, the correlation identifier from the original message is used.
	Otherwise, the message identifier is copied from the original message.

BackoutCount

Description:	Backout counter.
Data type:	MQLONG.
Value:	0.

ReplyToQ

Description:	Name of reply queue.
Data type:	MQCHAR48.
Values:	Blank.

ReplyToQMgr

Description:	Name of reply queue manager.
Data type:	MQCHAR48.
Value:	The queue manager name that generated the report message.

UserIdentifier

Description:	The user identifier of the application that generated the report message.
Data type:	MQCHAR12.
Value:	Copied from the original message descriptor.

AccountingToken

Description:	Accounting token that allows an application to charge for work done as a result of the message.
Data type:	MQBYTE32.
Value:	Copied from the original message descriptor.

ApplIdentityData

Description:	Application data relating to identity.
Data type:	MQCHAR32.
Values:	Copied from the original message descriptor.

PutApplType

Description:	Type of application that put the report message.
Data type:	MQLONG.
Value:	**MQAT_QMGR** Queue manager generated message.

PutApplName

Description:	Name of application that put the report message.
Data type:	MQCHAR28.
Value:	Either the first 28 bytes of the queue manager name, or the name of the MCA that generated the report message.

PutDate

Description:	Date when message was put.
Data type:	MQCHAR8.
Value:	As generated by the queue manager.

PutTime

Description:	Time when message was put.
Data type:	MQCHAR8.
Value:	As generated by the queue manager.

ApplOriginData

Description:	Application data relating to origin.
Data type:	MQCHAR4.
Value:	Blank.

If *Version* is MQMD_VERSION_2, the following additional fields are present:

GroupId

Description:	Identifies to which message group or logical message the physical message belongs.
Data type:	MQBYTE24.
Value:	Copied from the original message descriptor.

MsgSeqNumber

Description:	Sequence number of logical message within group.
Data type:	MQLONG.
Value:	Copied from the original message descriptor.

Offset

Description:	Offset of data in physical message from start of logical message.
Data type:	MQLONG.
Value:	Copied from the original message descriptor.

MsgFlags

Description:	Message flags that specify attributes of the message or control its processing.
Data type:	MQLONG.
Value:	Copied from the original message descriptor.

OriginalLength

Description:	Length of original message.
Data type:	MQLONG.
Value:	Copied from the original message descriptor.

Activity report MQEPH (Embedded PCF header)

Use this page to view the values contained by the MQEPH structure for an activity report

The MQEPH structure contains a description of both the PCF information that accompanies the message data of an activity report, and the application message data that follows it.

For an activity report, the MQEPH structure contains the following values:

StrucId

Description:	Structure identifier.
Data type:	MQCHAR4.
Value:	MQEPH_STRUC_ID.

Version

Description:	Structure version number.
Data type:	MQLONG.
Values:	MQEPH_VERSION_1.

StrucLength

Description:	Structure length.
Data type:	MQLONG.
Value:	Total length of the structure including the PCF parameter structures that follow it.

Encoding

Description:	Numeric encoding of the message data that follows the last PCF parameter structure.
Data type:	MQLONG.
Value:	If any data from the original application message data is included in the report message, the value will be copied from the *Encoding* field of the original message descriptor. Otherwise, 0.

CodedCharSetId

Description:	Character set identifier of the message data that follows the last PCF parameter structure.

Data type:	MQLONG.
Value:	If any data from the original application message data is included in the report message, the value will be copied from the *CodedCharSetId* field of the original message descriptor.
	Otherwise, MQCCSI_UNDEFINED.

Format

Description:	Format name of message data that follows the last PCF parameter structure.
Data type:	MQCHAR8.
Value:	If any data from the original application message data is included in the report message, the value will be copied from the *Format* field of the original message descriptor.
	Otherwise, MQFMT_NONE.

Flags

Description:	Flags that specify attributes of the structure or control its processing.
Data type:	MQLONG.
Value:	**MQEPH_CCSID_EMBEDDED** Specifies that the character set of the parameters containing character data is specified individually within the *CodedCharSetId* field in each structure.

PCFHeader

Description:	Programmable Command Format Header
Data type:	MQCFH.
Value:	See "Activity report MQCFH (PCF header)" on page 101.

Activity report MQCFH (PCF header)

Use this page to view the PCF values contained by the MQCFH structure for an activity report

For an activity report, the MQCFH structure contains the following values:

Type

Description:	Structure type that identifies the content of the report message.
Data type:	MQLONG.
Value:	**MQCFT_REPORT** Message is a report.

StrucLength

Description:	Structure length.
Data type:	MQLONG.
Value:	**MQCFH_STRUC_LENGTH** Length in bytes of MQCFH structure.

Version

Description:	Structure version number.

Data type:	MQLONG.
Values:	MQCFH_VERSION_3

Command

Description:	Command identifier. This identifies the category of the message.
Data type:	MQLONG.
Values:	**MQCMD_ACTIVITY_MSG**
	Message activity.

MsgSeqNumber

Description:	Message sequence number. This is the sequence number of the message within a group of related messages.
Data type:	MQLONG.
Values:	1.

Control

Description:	Control options.
Data type:	MQLONG.
Values:	MQCFC_LAST.

CompCode

Description:	Completion code.
Data type:	MQLONG.
Values:	MQCC_OK.

Reason

Description:	Reason code qualifying completion code.
Data type:	MQLONG.
Values:	MQRC_NONE.

ParameterCount

Description:	Count of parameter structures. This is the number of parameter structures that follow the MQCFH structure. A group structure (MQCFGR), and its included parameter structures, are counted as one structure only.
Data type:	MQLONG.
Values:	1 or greater.

Activity report message data

Use this page to view the parameters contained by the *Activity* PCF group in an activity report message. Some parameters are returned only when specific operations have been performed.

Activity report message data consists of the *Activity* PCF group and, if generated for a trace-route message, the *TraceRoute* PCF group. The *Activity* PCF group is detailed in this topic.

Some parameters, which are described as Operation-specific activity report message data, are returned only when specific operations have been performed.

For an activity report, the activity report message data contains the following parameters:

Activity

Description:	Grouped parameters describing the activity.
Identifier:	MQGACF_ACTIVITY.
Data type:	MQCFGR.
Included in PCF group:	None.
Parameters in PCF group:	*ActivityApplName* *ActivityApplType* *ActivityDescription* *Operation* *TraceRoute*
Returned:	Always.

ActivityApplName

Description:	Name of application that performed the activity.
Identifier:	MQCACF_APPL_NAME.
Data type:	MQCFST.
Included in PCF group:	*Activity*.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ActivityApplType

Description:	Type of application that performed the activity.
Identifier:	MQIA_APPL_TYPE.
Data type:	MQCFIN.
Included in PCF group:	*Activity*.
Returned:	Always.

ActivityDescription

Description:	Description of activity performed by the application.
Identifier:	MQCACF_ACTIVITY_DESCRIPTION.
Data type:	MQCFST.
Included in PCF group:	*Activity*.
Maximum length:	64
Returned:	Always.

Operation

Description:	Grouped parameters describing an operation of the activity.

Identifier:	MQGACF_OPERATION.
Data type:	MQCFGR.
Included in PCF group:	*Activity*.
Parameters in PCF group:	*OperationType* *OperationDate* *OperationTime* *Message* *QMgrName* *QSGName*

Note: Additional parameters are returned in this group depending on the operation type. These additional parameters are described as Operation-specific activity report message data.

Returned:	One *Operation* PCF group per operation in the activity.

OperationType

Description:	Type of operation performed.
Identifier:	MQIACF_OPERATION_TYPE.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Values:	MQOPER_*.
Returned:	Always.

OperationDate

Description:	Date when the operation was performed.
Identifier:	MQCACF_OPERATION_DATE.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_DATE_LENGTH.
Returned:	Always.

OperationTime

Description:	Time when the operation was performed.
Identifier:	MQCACF_OPERATION_TIME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_TIME_LENGTH.
Returned:	Always.

Message

Description:	Grouped parameters describing the message that caused the activity.
Identifier:	MQGACF_MESSAGE.
Data type:	MQCFGR.
Included in PCF group:	*Operation*.
Parameters in group:	*MsgLength* *MQMD* *EmbeddedMQMD*
Returned:	Always, except for Excluded Publish operations.

MsgLength

Description:	Length of the message that caused the activity, before the activity occurred.
Identifier:	MQIACF_MSG_LENGTH.
Data type:	MQCFIN.
Included in PCF group:	*Message*.
Returned:	Always.

MQMD

Description:	Grouped parameters related to the message descriptor of the message that caused the activity.
Identifier:	MQGACF_MQMD.
Data type:	MQCFGR.
Included in PCF group:	*Message*.

Parameters in group:	StrucId
	Version
	Report
	MsgType
	Expiry
	Feedback
	Encoding
	CodedCharSetId
	Format
	Priority
	Persistence
	MsgId
	CorrelId
	BackoutCount
	ReplyToQ
	ReplyToQMgr
	UserIdentifier
	AccountingToken
	ApplIdentityData
	PutApplType
	PutApplName
	PutDate
	PutTime
	ApplOriginData
	GroupId
	MsgSeqNumber
	Offset
	MsgFlags
	OriginalLength
Returned:	Always, except for Excluded Publish operations.

EmbeddedMQMD

Description:	Grouped parameters describing the message descriptor embedded within a message on a transmission queue.
Identifier:	MQGACF_EMBEDDDED_MQMD.
Data type:	MQCFGR.
Included in PCF group:	*Message.*

Parameters in group:	*StrucId*
	Version
	Report
	MsgType
	Expiry
	Feedback
	Encoding
	CodedCharSetId
	Format
	Priority
	Persistence
	MsgId
	CorrelId
	BackoutCount
	ReplyToQ
	ReplyToQMgr
	UserIdentifier
	AccountingToken
	ApplIdentityData
	PutApplType
	PutApplName
	PutDate
	PutTime
	ApplOriginData
	GroupId
	MsgSeqNumber
	Offset
	MsgFlags
	OriginalLength
Returned:	For Get operations where the queue resolves to a transmission queue.

StrucId

Description:	Structure identifier
Identifier:	MQCACF_STRUC_ID.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	4.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

Version

Description:	Structure version number.
Identifier:	MQIACF_VERSION.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.

Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

Report

Description:	Options for report messages.
Identifier:	MQIACF_REPORT.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

MsgType

Description:	Indicates type of message.
Identifier:	MQIACF_MSG_TYPE.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

Expiry

Description:	Message lifetime.
Identifier:	MQIACF_EXPIRY.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

Feedback

Description:	Feedback or reason code.
Identifier:	MQIACF_FEEDBACK.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

Encoding

Description:	Numeric encoding of message data.
Identifier:	MQIACF_ENCODING.
Data type:	MQCFIN.

Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

CodedCharSetId

Description:	Character set identifier of message data.
Identifier:	MQIA_CODED_CHAR_SET_ID.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

Format

Description:	Format name of message data
Identifier:	MQCACH_FORMAT_NAME.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_FORMAT_LENGTH.
Returned:	Always, except for Excluded Publish operations.

Priority

Description:	Message priority.
Identifier:	MQIACF_PRIORITY.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations.

Persistence

Description:	Message persistence.
Identifier:	MQIACF_PERSISTENCE.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations.

MsgId

Description:	Message identifier.
Identifier:	MQBACF_MSG_ID.
Data type:	MQCFBS.

Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_MSG_ID_LENGTH.
Returned:	Always, except for Excluded Publish operations.

CorrelId

Description:	Correlation identifier.
Identifier:	MQBACF_CORREL_ID.
Data type:	MQCFBS.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_CORREL_ID_LENGTH.
Returned:	Always, except for Excluded Publish operations.

BackoutCount

Description:	Backout counter.
Identifier:	MQIACF_BACKOUT_COUNT.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish operations and in MQMD for Publish and Discarded Publish operations.

ReplyToQ

Description:	Name of reply queue.
Identifier:	MQCACF_REPLY_TO_QUEUE.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish operations.

ReplyToQMgr

Description:	Name of reply queue manager.
Identifier:	MQCACF_REPLY_TO_Q_MGR.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

UserIdentifier

Description:	The user identifier of the application that originated the message.
Identifier:	MQCACF_USER_IDENTIFIER.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_USER_ID_LENGTH.
Returned:	Always, except for Excluded Publish Operations.

AccountingToken

Description:	Accounting token that allows an application to charge for work done as a result of the message.
Identifier:	MQBACF_ACCOUNTING_TOKEN.
Data type:	MQCFBS.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_ACCOUNTING_TOKEN_LENGTH.
Returned:	Always, except for Excluded Publish Operations.

ApplIdentityData

Description:	Application data relating to identity.
Identifier:	MQCACF_APPL_IDENTITY_DATA.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_APPL_IDENTITY_DATA_LENGTH.
Returned:	Always, except for Excluded Publish Operations.

PutApplType

Description:	Type of application that put the message.
Identifier:	MQIA_APPL_TYPE.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

PutApplName

Description:	Name of application that put the message.
Identifier:	MQCACF_APPL_NAME.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.

Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

PutDate

Description:	Date when message was put.
Identifier:	MQCACF_PUT_DATE.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_PUT_DATE_LENGTH.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

PutTime

Description:	Time when message was put.
Identifier:	MQCACF_PUT_TIME.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_PUT_TIME_LENGTH.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

ApplOriginData

Description:	Application data relating to origin.
Identifier:	MQCACF_APPL_ORIGIN_DATA.
Data type:	MQCFST.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_APPL_ORIGIN_DATA_LENGTH.
Returned:	Always, except for Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

GroupId

Description:	Identifies to which message group or logical message the physical message belongs.
Identifier:	MQBACF_GROUP_ID.
Data type:	MQCFBS.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Maximum length:	MQ_GROUP_ID_LENGTH.
Returned:	If the *Version* is specified as MQMD_VERSION_2. Not returned in Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

MsgSeqNumber

Description:	Sequence number of logical message within group.
Identifier:	MQIACH_MSG_SEQUENCE_NUMBER.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	If *Version* is specified as MQMD_VERSION_2. Not returned in Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

Offset

Description:	Offset of data in physical message from start of logical message.
Identifier:	MQIACF_OFFSET.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	If *Version* is specified as MQMD_VERSION_2. Not returned in Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

MsgFlags

Description:	Message flags that specify attributes of the message or control its processing.
Identifier:	MQIACF_MSG_FLAGS.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	If *Version* is specified as MQMD_VERSION_2. Not returned in Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

OriginalLength

Description:	Length of original message.
Identifier:	MQIACF_ORIGINAL_LENGTH.
Data type:	MQCFIN.
Included in PCF group:	*MQMD* or *EmbeddedMQMD*.
Returned:	If *Version* is specified as MQMD_VERSION_2. Not returned in Excluded Publish Operations and in MQMD for Publish and Discarded Publish Operations.

QMgrName

Description:	Name of the queue manager where the activity was performed.
Identifier:	MQCA_Q_MGR_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_MGR_NAME_LENGTH

Returned:	Always.

QSGName

Description:	Name of the queue-sharing group to which the queue manager where the activity was performed belongs.
Identifier:	MQCA_QSG_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_QSG_NAME_LENGTH
Returned:	If the activity was performed on a WebSphere MQ for z/OS queue manager.

TraceRoute

Description:	Grouped parameters specifying attributes of the trace-route message.
Identifier:	MQGACF_TRACE_ROUTE.
Data type:	MQCFGR.
Contained in PCF group:	*Activity*.

Parameters in group:	`Detail`
	`RecordedActivities`
	`UnrecordedActivities`
	`DiscontinuityCount`
	`MaxActivities`
	`Accumulate`
	`Forward`
	`Deliver`

Returned:	If the activity was performed on behalf of the trace-route message.

The values of the parameters in the *TraceRoute* PCF group are those from the trace-route message at the time the activity report was generated.

Operation-specific activity report message data

Use this page to view the additional PCF parameters that might be returned in the PCF group *Operation* in an activity report, depending on the value of the *OperationType* parameter

The additional parameters vary depending on the following operation types:

Get/Browse (MQOPER_GET/MQOPER_BROWSE)

The additional activity report message data parameters that are returned in the PCF group *Operation* for the Get/Browse (MQOPER_GET/MQOPER_BROWSE) operation type (a message on a queue was got, or browsed).

QName

Description:	The name of the queue that was opened.
Identifier:	MQCA_Q_NAME.
Data type:	MQCFST.

Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_NAME_LENGTH
Returned:	Always.

ResolvedQName

Description:	The name that the opened queue resolves to.
Identifier:	MQCACF_RESOLVED_Q_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_NAME_LENGTH
Returned:	Always.

Discard (MQOPER_DISCARD)

The additional activity report message data parameters that are returned in the PCF group *Operation* for the Discard (MQOPER_DISCARD) operation type (a message was discarded).

Feedback

Description:	The reason for the message being discarded.
Identifier:	MQIACF_FEEDBACK.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Returned:	Always.

QName

Description:	The name of the queue that was opened.
Identifier:	MQCA_Q_NAME.
Data type:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH
Included in PCF group:	*Operation*.
Returned:	If the message was discarded because it was unsuccessfully put to a queue.

RemoteQMgrName

Description:	The name of the queue manager to which the message was destined.
Identifier:	MQCA_REMOTE_Q_MGR_NAME.
Data type:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Included in PCF group:	*Operation*.
Returned:	If the value of *Feedback* is MQFB_NOT_FORWARDED.

Publish/Discarded Publish/Excluded Publish (MQOPER_PUBLISH/ MQOPER_DISCARDED_PUBLISH/MQOPER_EXCLUDED_PUBLISH)

The additional activity report message data parameters that are returned in the PCF group *Operation* for the Publish/Discarded Publish/Excluded Publish (MQOPER_PUBLISH/MQOPER_DISCARDED_PUBLISH/ MQOPER_EXCLUDED_PUBLISH) operation type (a publish/subscribe message was delivered, discarded, or excluded).

SubId

Description:	The subscription identifier.
Identifier:	MQBACF_SUB_ID.
Data type:	MQCFBS.
Included in PCF group:	*Operation*.
Returned:	Always.

SubLevel

Description:	The subscription level.
Identifier:	MQIACF_SUB_LEVEL.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Returned:	Always.

Feedback

Description:	The reason for discarding the message.
Identifier:	MQIACF_FEEDBACK.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Returned:	If the message was discarded because it was not delivered to a subscriber, or the message was not delivered because the subscriber was excluded.

The Publish operation MQOPER_PUBLISH provides information about a message delivered to a particular subscriber. This operation describes the elements of the onward message that might have changed from the message described in the associated Put operation. Similarly to a Put operation, it contains a message group MQGACF_MESSAGE and, inside that, an MQMD group MQGACF_MQMD. However, this MQMD group contains only the following fields, which can be overridden by a subscriber: *Format, Priority, Persistence, MsgId, CorrelId, UserIdentifier, AccountingToken, ApplIdentityData.*

The *SubId* and *SubLevel* of the subscriber are included in the operation information. You can use the *SubID* with the MQCMD_INQUIRE_SUBSCRIBER PCF command to retrieve all other attributes for a subscriber.

The Discarded Publish operation MQOPER_DISCARDED_PUBLISH is analogous to the Discard operation that is used when a message is not delivered in point-to-point messaging. A message is not delivered to a subscriber if the message was explicitly requested not to be delivered to a local destination and this subscriber specifies a local destination. A message is also considered not delivered if there is a problem getting the message to the destination queue, for example, because the queue is full.

The information in a Discarded Publish operation is the same as for a Publish operation, with the addition of a *Feedback* field that gives the reasons why the message was not delivered. This feedback field contains MQFB_* or MQRC_* values that are common with the MQOPER_DISCARD operation. The reason for discarding a publish, as opposed to excluding it, are the same as the reasons for discarding a put.

The Excluded Publish operation MQOPER_EXCLUDED_PUBLISH provides information about a subscriber that was considered for delivery of the message, because the topic on which the subscriber is subscribing matches that of the associated Put operation, but the message was not delivered to the subscriber because other selection criteria do not match with the message that is being put to the topic. As with a Discarded Publish operation, the *Feedback* field provides information about the reason why this subscription was excluded. However, unlike the Discarded Publish operation, no message-related information is provided because no message was generated for this subscriber.

Put/Put Reply/Put Report (MQOPER_PUT/MQOPER_PUT_REPLY/ MQOPER_PUT_REPORT)

The additional activity report message data parameters that are returned in the PCF group *Operation* for the Put/Put Reply/Put Report (MQOPER_PUT/MQOPER_PUT_REPLY/MQOPER_PUT_REPORT) operation type (a message, reply message, or report message was put to a queue).

QName

Description:	The name of the queue that was opened.
Identifier:	MQCA_Q_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_NAME_LENGTH
Returned:	Always, apart from one exception: not returned if the Put operation is to a topic, contained within a publish activity.

ResolvedQName

Description:	The name that the opened queue resolves to.
Identifier:	MQCACF_RESOLVED_Q_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_NAME_LENGTH
Returned:	When the opened queue could be resolved. Not returned if the Put operation is to a topic, contained within a publish activity.

RemoteQName

Description:	The name of the opened queue, as it is known on the remote queue manager.
Identifier:	MQCA_REMOTE_Q_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_NAME_LENGTH

Returned:	If the opened queue is a remote queue. Not returned if the Put operation is to a topic, contained within a publish activity.

RemoteQMgrName

Description:	The name of the remote queue manager on which the remote queue is defined.
Identifier:	MQCA_REMOTE_Q_MGR_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Returned:	If the opened queue is a remote queue. Not returned if the Put operation is to a topic, contained within a publish activity.

TopicString

Description:	The full topic string to which the message is being put.
Identifier:	MQCA_TOPIC_STRING.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Returned:	If the Put operation is to a topic, contained within a publish activity.

Feedback

Description:	The reason for the message being put on the dead-letter queue.
Identifier:	MQIACF_FEEDBACK.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Returned:	If the message was put on the dead-letter queue.

Receive (MQOPER_RECEIVE)

The additional activity report message data parameters that are returned in the PCF group *Operation* for the Receive (MQOPER_RECEIVE) operation type (a message was received on a channel).

ChannelName

Description:	The name of the channel on which the message was received.
Identifier:	MQCACH_CHANNEL_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_CHANNEL_NAME_LENGTH
Returned:	Always.

ChannelType

Description:	The type of channel on which the message was received.

Identifier:	MQIACH_CHANNEL_TYPE.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Returned:	Always.

RemoteQMgrName

Description:	The name of the queue manager from which the message was received.
Identifier:	MQCA_REMOTE_Q_MGR_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Returned:	Always.

Send (MQOPER_SEND)

The additional activity report message data parameters that are returned in the PCF group *Operation* for the Send (MQOPER_SEND) operation type (a message was sent on a channel).

ChannelName

Description:	The name of the channel where the message was sent.
Identifier:	MQCACH_CHANNEL_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

ChannelType

Description:	The type of channel where the message was sent.
Identifier:	MQIACH_CHANNEL_TYPE.
Data type:	MQCFIN.
Included in PCF group:	*Operation*.
Returned:	Always.

XmitQName

Description:	The transmission queue from which the message was retrieved.
Identifier:	MQCACH_XMIT_Q_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_NAME_LENGTH.

| Returned: | Always. |

RemoteQMgrName

Description:	The name of the remote queue manager to which the message was sent.
Identifier:	MQCA_REMOTE_Q_MGR_NAME.
Data type:	MQCFST.
Included in PCF group:	*Operation*.
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Returned:	Always.

Trace-route message reference

Use this page to obtain an overview of the trace-route message format. The trace-route message data includes parameters that describe the activities that the trace-route message has caused

Trace-route message format

Trace-route messages are standard WebSphere MQ messages containing a message descriptor and message data. The message data contains information about the activities performed on a trace-route message as it has been routed through a queue manager network.

Trace-route messages contain the following information:

A message descriptor
An MQMD structure, with the *Format* field set to MQFMT_ADMIN or MQFMT_EMBEDDED_PCF.

Message data
Consists of either:

- A PCF header (MQCFH) and trace-route message data, if *Format* is set to MQFMT_ADMIN, or
- An embedded PCF header (MQEPH), trace-route message data, and additional user-specified message data, if *Format* is set to MQFMT_EMBEDDED_PCF.

When using the WebSphere MQ display route application to generate a trace-route message, *Format* is set to MQFMT_ADMIN.

The content of the trace-route message data is determined by the *Accumulate* parameter from the *TraceRoute* PCF group, as follows:

- If *Accumulate* is set to MQROUTE_ACCUMULATE_NONE, the trace-route message data contains the *TraceRoute* PCF group.
- If *Accumulate* is set to either MQROUTE_ACCUMULATE_IN_MSG or MQROUTE_ACCUMULATE_AND_REPLY, the trace-route message data contains the *TraceRoute* PCF group and zero or more *Activity* PCF groups.

shows the structure of a trace-route message.

Table 20. Trace-route message format		
MQMD structure	**Embedded PCF header MQEPH structure**	**Trace-route message data**

Structure identifier
Structure version
Report options
Message type
Expiration time
Feedback
Encoding
Coded character set ID
Message format
Priority
Persistence
Message identifier
Correlation identifier
Backout count
Reply-to queue
Reply-to queue manager
User identifier
Accounting token
Application identity data
Application type
Application name
Put date
Put time
Application origin data
Group identifier
Message sequence number
Offset
Message flags
Original length
	Structure identifier
	Structure version
	Structure length
	Encoding
	Coded character set ID
	Message format
	Flags
	PCF header (MQCFH)
	  Structure type
	  Structure length
	  Structure version
	  Command identifier
	  Message sequence number
	  Control options
	  Completion code
	  Reason code
	  Parameter count
		TraceRoute
		  Detail
		  Recorded activities
		  Unrecorded activities
		  Discontinuity count
		  Max activities
		  Accumulate
		  Deliver

Trace-route message MQMD (message descriptor)

Use this page to view the values contained by the MQMD structure for a trace-route message

StrucId

Description:	Structure identifier.
Data type:	MQCHAR4.
Value:	MQMD_STRUC_ID.

Version

Description:	Structure version number.
Data type:	MQLONG.
Values:	**MQMD_VERSION_1.**

Report

Description:	Options for report messages.
Data type:	MQLONG.

Value:	Set according to requirements. Common report options follow:

MQRO_DISCARD_MSG
  The message is discarded on arrival to a local queue.

MQRO_PASS_DISCARD_AND_EXPIRY
  Every response (activity reports or trace-route reply message) will have the report option MQRO_DISCARD_MSG set, and the remaining expiry passed on. This ensures that responses do not remain in the queue manager network indefinitely.

MsgType

Description:	Type of message.
Data type:	MQLONG.
Value:	If the *Accumulate* parameter in the TraceRoute group is specified as MQROUTE_ACCUMULATE_AND_REPLY, then message type is MQMT_REQUEST
	Otherwise:
	MQMT_DATAGRAM.

Expiry

Description:	Message lifetime.
Data type:	MQLONG.
Value:	Set according to requirements. This parameter can be used to ensure trace-route messages are not left in a queue manager network indefinitely.

Feedback

Description:	Feedback or reason code.
Data type:	MQLONG.
Value:	**MQFB_NONE.**

Encoding

Description:	Numeric encoding of message data.
Data type:	MQLONG.
Value:	Set as appropriate.

CodedCharSetId

Description:	Character set identifier of message data.
Data type:	MQLONG.
Value:	Set as appropriate.

Format

Description:	Format name of message data
Data type:	MQCHAR8.

Value:	**MQFMT_ADMIN**	
	Admin message. No user data follows the *TraceRoute* PCF group.	
	MQFMT_EMBEDDED_PCF	
	Embedded PCF message. User data follows the *TraceRoute* PCF group.	

Priority

Description:	Message priority.
Data type:	MQLONG.
Value:	Set according to requirements.

Persistence

Description:	Message persistence.
Data type:	MQLONG.
Value:	Set according to requirements.

MsgId

Description:	Message identifier.
Data type:	MQBYTE24.
Value:	Set according to requirements.

CorrelId

Description:	Correlation identifier.
Data type:	MQBYTE24.
Value:	Set according to requirements.

BackoutCount

Description:	Backout counter.
Data type:	MQLONG.
Value:	0.

ReplyToQ

Description:	Name of reply queue.
Data type:	MQCHAR48.
Values:	Set according to requirements.
	If *MsgType* is set to MQMT_REQUEST or if *Report* has any report generating options set, then this parameter must be non-blank.

ReplyToQMgr

Description:	Name of reply queue manager.
Data type:	MQCHAR48.
Value:	Set according to requirements.

UserIdentifier

Description:	The user identifier of the application that originated the message.
Data type:	MQCHAR12.
Value:	Set as normal.

AccountingToken

Description:	Accounting token that allows an application to charge for work done as a result of the message.
Data type:	MQBYTE32.
Value:	Set as normal.

ApplIdentityData

Description:	Application data relating to identity.
Data type:	MQCHAR32.
Values:	Set as normal.

PutApplType

Description:	Type of application that put the message.
Data type:	MQLONG.
Value:	Set as normal.

PutApplName

Description:	Name of application that put the message.
Data type:	MQCHAR28.
Value:	Set as normal.

PutDate

Description:	Date when message was put.
Data type:	MQCHAR8.
Value:	Set as normal.

PutTime

Description:	Time when message was put.
Data type:	MQCHAR8.
Value:	Set as normal.

ApplOriginData

Description:	Application data relating to origin.
Data type:	MQCHAR4.
Value:	Set as normal..

Trace-route message MQEPH (Embedded PCF header)

Use this page to view the values contained by the MQEPH structure for a trace-route message

The MQEPH structure contains a description of both the PCF information that accompanies the message data of a trace-route message, and the application message data that follows it. An MQEPH structure is used only if additional user message data follows the TraceRoute PCF group.

For a trace-route message, the MQEPH structure contains the following values:

StrucId

Description:	Structure identifier.
Data type:	MQCHAR4.
Value:	MQEPH_STRUC_ID.

Version

Description:	Structure version number.
Data type:	MQLONG.
Values:	MQEPH_VERSION_1.

StrucLength

Description:	Structure length.
Data type:	MQLONG.
Value:	Total length of the structure including the PCF parameter structures that follow it.

Encoding

Description:	Numeric encoding of the message data that follows the last PCF parameter structure.
Data type:	MQLONG.
Value:	The encoding of the message data.

CodedCharSetId

Description:	Character set identifier of the message data that follows the last PCF parameter structure.
Data type:	MQLONG.
Value:	The character set of the message data.

Format

Description:	Format name of the message data that follows the last PCF parameter structure.
Data type:	MQCHAR8.
Value:	The format name of the message data.

Flags

Description:	Flags that specify attributes of the structure or control its processing.
Data type:	MQLONG.

Value:

MQEPH_NONE
No flags specified.

MQEPH_CCSID_EMBEDDED
Specifies that the character set of the parameters containing character data is specified individually within the *CodedCharSetId* field in each structure.

PCFHeader

Description:	Programmable Command Format Header
Data type:	MQCFH.
Value:	See "Trace-route message MQCFH (PCF header)" on page 126.

Trace-route message MQCFH (PCF header)

Use this page to view the PCF values contained by the MQCFH structure for a trace-route message

For a trace-route message, the MQCFH structure contains the following values:

Type

Description:	Structure type that identifies the content of the message.
Data type:	MQLONG.

Value:

MQCFT_TRACE_ROUTE
Message is a trace-route message.

StrucLength

Description:	Structure length.
Data type:	MQLONG.

Value:

MQCFH_STRUC_LENGTH
Length in bytes of MQCFH structure.

Version

Description:	Structure version number.
Data type:	MQLONG.
Values:	MQCFH_VERSION_3

Command

Description:	Command identifier. This identifies the category of the message.
Data type:	MQLONG.

Values:

MQCMD_TRACE_ROUTE
Trace-route message.

MsgSeqNumber

Description:	Message sequence number. This is the sequence number of the message within a group of related messages.
Data type:	MQLONG.
Values:	1.

Control

Description:	Control options.
Data type:	MQLONG.
Values:	MQCFC_LAST.

CompCode

Description:	Completion code.
Data type:	MQLONG.
Values:	MQCC_OK.

Reason

Description:	Reason code qualifying completion code.
Data type:	MQLONG.
Values:	MQRC_NONE.

ParameterCount

Description:	Count of parameter structures. This is the number of parameter structures that follow the MQCFH structure. A group structure (MQCFGR), and its included parameter structures, are counted as one structure only.
Data type:	MQLONG.
Values:	1 or greater.

Trace-route message data

Use this page to view the parameters that make up the *TraceRoute* PCF group part of trace-route message data

The content of trace-route message data depends on the *Accumulate* parameter from the *TraceRoute* PCF group. Trace-route message data consists of the *TraceRoute* PCF group, and zero or more *Activity* PCF groups. The *TraceRoute* PCF group is detailed in this topic. Refer to the related information for details of the *Activity* PCF group.

Trace-route message data contains the following parameters:

TraceRoute

Description:	Grouped parameters specifying attributes of the trace-route message. For a trace-route message, some of these parameters can be altered to control how it is processed.
Identifier:	MQGACF_TRACE_ROUTE.
Data type:	MQCFGR.
Contained in PCF group:	None.

Parameters in group:	*Detail*
	RecordedActivities
	UnrecordedActivities
	DiscontinuityCount
	MaxActivities
	Accumulate
	Forward
	Deliver

Detail

Description:	The detail level that will be recorded for the activity.
Identifier:	MQIACF_ROUTE_DETAIL.
Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.
Values:	**MQROUTE_DETAIL_LOW** Activities performed by user-written application are recorded. **MQROUTE_DETAIL_MEDIUM** Activities specified in MQROUTE_DETAIL_LOW are recorded. Additionally, activities performed by MCAs are recorded. **MQROUTE_DETAIL_HIGH** Activities specified in MQROUTE_DETAIL_LOW, and MQROUTE_DETAIL_MEDIUM are recorded. MCAs do not record any further activity information at this level of detail. This option is only available to user-written applications that are to record further activity information.

RecordedActivities

Description:	The number of activities that the trace-route message has caused, where information was recorded.
Identifier:	MQIACF_RECORDED_ACTIVITIES.
Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.

UnrecordedActivities

Description:	The number of activities that the trace-route message has caused, where information was not recorded.
Identifier:	MQIACF_UNRECORDED_ACTIVITIES.
Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.

DiscontinuityCount

Description:	The number of times a trace-route message has been received from a queue manager that does not support trace-route messaging.
Identifier:	MQIACF_DISCONTINUITY_COUNT.

Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.

MaxActivities

Description:	The maximum number of activities the trace-route message can be involved in before it stops being processed.
Identifier:	MQIACF_MAX_ACTIVITIES.
Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.
Value:	**A positive integer** The maximum number of activities. **MQROUTE_UNLIMITED_ACTIVITIES** An unlimited number of activities.

Accumulate

Description:	Specifies whether activity information is accumulated within the trace-route message, and whether a reply message containing the accumulated activity information is generated before the trace-route message is discarded or is put on a non-transmission queue.
Identifier:	MQIACF_ROUTE_ACCUMULATION.
Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.
Value:	**MQROUTE_ACCUMULATE_NONE** Activity information is not accumulated in the message data of the trace-route message. **MQROUTE_ACCUMULATE_IN_MSG** Activity information is accumulated in the message data of the trace-route message. **MQROUTE_ACCUMULATE_AND_REPLY** Activity information is accumulated in the message data of the trace-route message, and a trace-route reply message will be generated.

Forward

Description:	Specifies queue managers that the trace-route message can be forwarded to. When determining whether to forward a message to a remote queue manager, queue managers use the algorithm that is described in Forwarding.
Identifier:	MQIACF_ROUTE_FORWARDING.
Data type:	MQCFIN.
Contained in PCF group:	*TraceRoute*.

Value: **MQROUTE_FORWARD_IF_SUPPORTED**
                The trace-route message is only forwarded to queue managers that will
                honor the value of the *Deliver* parameter from the *TraceRoute* group.

                MQROUTE_FORWARD_ALL
                The trace-route message is forwarded to any queue manager, regardless of
                whether the value of the *Deliver* parameter will be honored.

Deliver

Description: Specifies the action to be taken if the trace-route message arrives at the
                destination queue successfully.

Identifier: MQIACF_ROUTE_DELIVERY.

Data type: MQCFIN.

Contained in PCF *TraceRoute*.
group:

Value: **MQROUTE_DELIVER_YES**
                On arrival, the trace-route message is put on the target queue. Any
                application performing a destructive get on the target queue can receive
                the trace-route message.

                MQROUTE_DELIVER_NO
                On arrival, the trace-route message is discarded.

Trace-route reply message reference

Use this page to obtain an overview of the trace-route reply message format. The trace-route reply
message data is a duplicate of the trace-route message data from the trace-route message for which it
was generated

Trace-route reply message format

Trace-route reply messages are standard WebSphere MQ messages containing a message descriptor and
message data. The message data contains information about the activities performed on a trace-route
message as it has been routed through a queue manager network.

Trace-route reply messages contain the following information:

A message descriptor
    An MQMD structure

Message data
    A PCF header (MQCFH) and trace-route reply message data

Trace-route reply message data consists of one or more *Activity* PCF groups.

When a trace-route message reaches its target queue, a trace-route reply message can be generated that
contains a copy of the activity information from the trace-route message. The trace-route reply message
will be delivered to a reply-to queue or to a system queue.

Table 21 on page 131 shows the structure of a trace-route reply message, including parameters that are
only returned under certain conditions.

Table 21. Trace-route reply message format		
MQMD structure	**PCF header MQCFH structure**	**Trace-route reply message data**

Structure identifier
Structure version
Report options
Message type
Expiration time
Feedback
Encoding
Coded character set ID
Message format
Priority
Persistence
Message identifier
Correlation identifier
Backout count
Reply-to queue
Reply-to queue manager
User identifier
Accounting token
Application identity data
Application type
Application name
Put date
Put time
Application origin data
Group identifier
Message sequence number
Offset
Message flags
Original length
	PCF header (MQCFH)
	  Structure type
	  Structure length
	  Structure version
	  Command identifier
	  Message sequence number
	  Control options
	  Completion code
	  Reason code
	  Parameter count
		Activity
		  Activity application name
		  Activity application type
		  Activity description
		  Operation
		    Operation type
		    Operation date
		    Operation time
		    Message
		      Message length
		      MQMD
		      EmbeddedMQMD
		    Queue manager name
		    Queue sharing group name
		    Queue name [1] [2] [3]
		    Resolved queue name [1] [3]
		    Remote queue name [3]
		    Remote queue manager-
		      name [2] [3] [4] [5]
		    Feedback [2]
		    Channel name [4] [5]
		    Channel type [4] [5]
		    Transmission queue name [5]
		  TraceRoute
		    Detail
		    Recorded activities
		    Unrecorded activities
		    Discontinuity count
		    Max activities
		    Accumulate
		    Deliver

Note:

1. Returned for Get and Browse operations.

2. Returned for Discard operations.

3. Returned for Put, Put Reply, and Put Report operations.

4. Returned for Receive operations.

5. Returned for Send operations.

Trace-route reply message MQMD (message descriptor)

Use this page to view the values contained by the MQMD structure for a trace-route reply message

For a trace-route reply message, the MQMD structure contains the parameters described in Activity report message descriptor. Some of the parameter values in a trace-route reply message descriptor are different from those in an activity report message descriptor, as follows:

MsgType

        Description: Type of message.

        Data type: MQLONG.

Value:	**MQMT_REPLY**

Feedback

Description:	Feedback or reason code.
Data type:	MQLONG.
Value:	**MQFB_NONE**

Encoding

Description:	Numeric encoding of message data.
Data type:	MQLONG.
Value:	Copied from trace-route message descriptor.

CodedCharSetId

Description:	Character set identifier of message data.
Data type:	MQLONG.
Value:	Copied from trace-route message descriptor.

Format

Description:	Format name of message data
Data type:	MQCHAR8.
Value:	**MQFMT_ADMIN**
	Admin message.

Trace-route reply message MQCFH (PCF header)

Use this page to view the PCF values contained by the MQCFH structure for a trace-route reply message

The PCF header (MQCFH) for a trace-route reply message is the same as for a trace-route message.

Trace-route reply message data

The trace-route reply message data is a duplicate of the trace-route message data from the trace-route message for which it was generated

The trace-route reply message data contains one or more *Activity* groups. The parameters are described in "Activity report message data" on page 102.

Accounting and statistics messages

Queue managers generate accounting and statistics messages to record information about the MQI operations performed by IBM WebSphere MQ applications, or to record information about the activities occurring in an IBM WebSphere MQ system.

Accounting messages
Accounting messages are used to record information about the MQI operations performed by IBM WebSphere MQ applications, see "Accounting messages" on page 133.

Statistics messages
Statistics messages are used to record information about the activities occurring in an IBM WebSphere MQ system, see "Statistics messages" on page 136. Some activity recorded in statistics messages relates to internal queue manager operations.

Accounting and statistics messages are delivered to one of two system queues. User applications can retrieve the messages from these system queues and use the recorded information for various purposes:

- Account for application resource use.
- Record application activity.
- Capacity planning.
- Detect problems in your queue manager network.
- Assist in determining the causes of problems in your queue manager network.
- Improve the efficiency of your queue manager network.
- Familiarize yourself with the running of your queue manager network.
- Confirm that your queue manager network is running correctly.

Accounting messages

Accounting messages record information about the MQI operations performed by WebSphere MQ applications. An accounting message is a PCF message that contains a number of PCF structures.

When an application disconnects from a queue manager, an accounting message is generated and delivered to the system accounting queue (SYSTEM.ADMIN.ACCOUNTING.QUEUE). For long running WebSphere MQ applications, intermediate accounting messages are generated as follows:

- When the time since the connection was established exceeds the configured interval.
- When the time since the last intermediate accounting message exceeds the configured interval.

Accounting messages are in the following categories:

MQI accounting messages
MQI accounting messages contain information relating to the number of MQI calls made using a connection to a queue manager.

Queue accounting messages
Queue accounting messages contain information relating to the number of MQI calls made using connections to a queue manager, grouped by queue.

Each queue accounting message can contain up to 100 records, with every record relating to an activity performed by the application with respect to a specific queue.

Accounting messages are recorded only for local queues. If an application makes an MQI call against an alias queue, the accounting data is recorded against the base queue, and, for a remote queue, the accounting data is recorded against the transmission queue.

Related reference
Use this page to view the structure of an MQI accounting message

Use this page to view the structure of a queue accounting message

Accounting message format

Accounting messages comprise a set of PCF fields that consist of a message descriptor and message data.

Message descriptor

- An accounting message MQMD (message descriptor)

Accounting message data

- An accounting message MQCFH (PCF header)
- Accounting message data that is always returned
- Accounting message data that is returned if available

The accounting message MQCFH (PCF header) contains information about the application, and the interval for which the accounting data was recorded.

Accounting message data comprises PCF parameters that store the accounting information. The content of accounting messages depends on the message category as follows:

MQI accounting message
MQI accounting message data consists of a number of PCF parameters, but no PCF groups.

Queue accounting message
Queue accounting message data consists of a number of PCF parameters, and in the range 1 through 100 *QAccountingData* PCF groups.

There is one *QAccountingData* PCF group for every queue that had accounting data collected. If an application accesses more than 100 queues, multiple accounting messages are generated. Each message has the *SeqNumber* in the MQCFH (PCF header) updated accordingly, and the last message in the sequence has the *Control* parameter in the MQCFH specified as MQCFC_LAST.

Accounting information collection

Use queue and queue manager attributes to control the collection of accounting information. You can also use MQCONNX options to control collection at the connection level.

MQI accounting information
Use the queue manager attribute ACCTMQI to control the collection of MQI accounting information

To change the value of this attribute, use the MQSC command, ALTER QMGR, and specify the parameter ACCTMQI. Accounting messages are generated only for connections that begin after accounting is enabled. The ACCTMQI parameter can have the following values:

ON
MQI accounting information is collected for every connection to the queue manager.

OFF
MQI accounting information is not collected. This is the default value.

For example, to enable MQI accounting information collection use the following MQSC command:

```
ALTER QMGR ACCTMQI(ON)
```

Queue accounting information
Use the queue attribute ACCTQ and the queue manager attribute ACCTQ to control the collection of queue accounting information.

To change the value of the queue attribute, use the MQSC command, ALTER QLOCAL and specify the parameter ACCTQ. Accounting messages are generated only for connections that begin after accounting is enabled. The queue attribute ACCTQ can have the following values:

ON
Queue accounting information for this queue is collected for every connection to the queue manager that opens the queue.

OFF
Queue accounting information for this queue is not collected.

QMGR
The collection of queue accounting information for this queue is controlled according to the value of the queue manager attribute ACCTQ. This is the default value.

To change the value of the queue manager attribute, use the MQSC command, ALTER QMGR and specify the parameter ACCTQ. The queue manager attribute ACCTQ can have the following values:

ON
Queue accounting information is collected for queues that have the queue attribute ACCTQ set as QMGR.

OFF

Queue accounting information is not collected for queues that have the queue attribute ACCTQ set as QMGR. This is the default value.

NONE

The collection of queue accounting information is disabled for all queues, regardless of the queue attribute ACCTQ.

If the queue manager attribute, ACCTQ, is set to NONE, the collection of queue accounting information is disabled for all queues, regardless of the queue attribute ACCTQ.

For example, to enable accounting information collection for the queue, Q1, use the following MQSC command:

```
ALTER QLOCAL(Q1) ACCTQ(ON)
```

To enable accounting information collection for all queues that specify the queue attribute ACCTQ as QMGR, use the following MQSC command:

```
ALTER QMGR ACCTQ(ON)
```

MQCONNX options

Use the **ConnectOpts** parameter on the MQCONNX call to modify the collection of both MQI and queue accounting information at the connection level by overriding the effective values of the queue manager attributes ACCTMQI and ACCTQ

The **ConnectOpts** parameter can have the following values:

MQCNO_ACCOUNTING_MQI_ENABLED

If the value of the queue manager attribute ACCTMQI is specified as OFF, MQI accounting is enabled for this connection. This is equivalent of the queue manager attribute ACCTMQI being specified as ON.

If the value of the queue manager attribute ACCTMQI is not specified as OFF, this attribute has no effect.

MQCNO_ACCOUNTING_MQI_DISABLED

If the value of the queue manager attribute ACCTMQI is specified as ON, MQI accounting is disabled for this connection. This is equivalent of the queue manager attribute ACCTMQI being specified as OFF.

If the value of the queue manager attribute ACCTMQI is not specified as ON, this attribute has no effect.

MQCNO_ACCOUNTING_Q_ENABLED

If the value of the queue manager attribute ACCTQ is specified as OFF, queue accounting is enabled for this connection. All queues with ACCTQ specified as QMGR, are enabled for queue accounting. This is equivalent of the queue manager attribute ACCTQ being specified as ON.

If the value of the queue manager attribute ACCTQ is not specified as OFF, this attribute has no effect.

MQCNO_ACCOUNTING_Q_DISABLED

If the value of the queue manager attribute ACCTQ is specified as ON, queue accounting is disabled for this connection. This is equivalent of the queue manager attribute ACCTQ being specified as OFF.

If the value of the queue manager attribute ACCTQ is not specified as ON, this attribute has no effect.

These overrides are by disabled by default. To enable them, set the queue manager attribute ACCTCONO to ENABLED. To enable accounting overrides for individual connections use the following MQSC command:

```
ALTER QMGR ACCTCONO(ENABLED)
```

Accounting message generation

Accounting messages are generated when an application disconnects from the queue manager. Intermediate accounting messages are also written for long running WebSphere MQ applications.

Accounting messages are generated in either of the following ways when an application disconnects:

- The application issues an MQDISC call
- The queue manager recognises that the application has terminated

Intermediate accounting messages are written for long running WebSphere MQ applications when the interval since the connection was established or since the last intermediate accounting message that was written exceeds the configured interval. The queue manager attribute, ACCTINT, specifies the time, in seconds, after which intermediate accounting messages can be automatically written. Accounting messages are generated only when the application interacts with the queue manager, so applications that remain connected to the queue manager for long periods without executing MQI requests do not generate accounting messages until the execution of the first MQI request following the completion of the accounting interval.

The default accounting interval is 1800 seconds (30 minutes). For example, to change the accounting interval to 900 seconds (15 minutes) use the following MQSC command:

```
ALTER QMGR ACCTINT(900)
```

Statistics messages

Statistics messages record information about the activities occurring in a WebSphere MQ system. An statistics messages is a PCF message that contains a number of PCF structures.

Statistics messages are delivered to the system queue (SYSTEM.ADMIN.STATISTICS.QUEUE) at configured intervals, whenever there is some activity.

Statistics messages are in the following categories:

MQI statistics messages
MQI statistics messages contain information relating to the number of MQI calls made during a configured interval. For example, the information can include the number of MQI calls issued by a queue manager.

Queue statistics messages
Queue statistics messages contain information relating to the activity of a queue during a configured interval. The information includes the number of messages put on, and retrieved from, the queue, and the total number of bytes processed by a queue.

Each queue statistics message can contain up to 100 records, with each record relating to the activity per queue for which statistics were collected.

Statistics messages are recorded only for local queues. If an application makes an MQI call against an alias queue, the statistics data is recorded against the base queue, and, for a remote queue, the statistics data is recorded against the transmission queue.

Channel statistics messages
Channel statistics messages contain information relating to the activity of a channel during a configured interval. For example the information might be the number of messages transferred by the channel, or the number of bytes transferred by the channel.

Each channel statistics message contains up to 100 records, with each record relating to the activity per channel for which statistics were collected.

Related reference
"MQI statistics information" on page 137
Use the queue manager attribute STATMQI to control the collection of MQI statistics information

"Queue statistics information" on page 138

Use the queue attribute STATQ and the queue manager attribute STATQ to control the collection of queue statistics information

Use the channel attribute STATCHL to control the collection of channel statistics information. You can also set queue manager attributes to control information collection. These attributes are available on distributed platforms and on IBM i.

Statistics messages format

Statistics messages comprise a set of PCF fields that consist of a message descriptor and message data.

Message descriptor

- A statistics message MQMD (message descriptor)

Accounting message data

- A statistics message MQCFH (PCF header)
- Statistics message data that is always returned
- Statistics message data that is returned if available

The statistics message MQCFH (PCF header) contains information about the interval for which the statistics data was recorded.

Statistics message data comprises PCF parameters that store the statistics information. The content of statistics messages depends on the message category as follows:

MQI statistics message
MQI statistics message data consists of a number of PCF parameters, but no PCF groups.

Queue statistics message
Queue statistics message data consists of a number of PCF parameters, and in the range 1 through 100 *QStatisticsData* PCF groups.

There is one *QStatisticsData* PCF group for every queue was active in the interval. If more than 100 queues were active in the interval, multiple statistics messages are generated. Each message has the *SeqNumber* in the MQCFH (PCF header) updated accordingly, and the last message in the sequence has the *Control* parameter in the MQCFH specified as MQCFC_LAST.

Channel statistics message
Channel statistics message data consists of a number of PCF parameters, and in the range 1 through 100 *ChlStatisticsData* PCF groups.

There is one *ChlStatisticsData* PCF group for every channel that was active in the interval. If more than 100 channels were active in the interval, multiple statistics messages are generated. Each message has the *SeqNumber* in the MQCFH (PCF header) updated accordingly, and the last message in the sequence has the *Control* parameter in the MQCFH specified as MQCFC_LAST.

Statistics information collection

Use queue, queue manager, and channel attributes to control the collection of statistics information

MQI statistics information
Use the queue manager attribute STATMQI to control the collection of MQI statistics information

To change the value of this attribute, use the MQSC command, ALTER QMGR and specify the parameter STATMQI. Statistics messages are generated only for queues that are opened after statistics collection has been enabled. The STATMQI parameter can have the following values:

ON
MQI statistics information is collected for every connection to the queue manager.

OFF
MQI statistics information is not collected. This is the default value.

For example, to enable MQI statistics information collection use the following MQSC command:

```
ALTER QMGR STATMQI(ON)
```

Queue statistics information

Use the queue attribute STATQ and the queue manager attribute STATQ to control the collection of queue statistics information

You can enable or disable queue statistics information collection for individual queues or for multiple queues. To control individual queues, set the queue attribute STATQ. You enable or disable queue statistics information collection at the queue manager level by using the queue manager attribute STATQ. For all queues that have the queue attribute STATQ specified with the value QMGR, queue statistics information collection is controlled at the queue manager level.

Queue statistics are incremented only for operations using IBM WebSphere MQ MQI Object Handles that were opened after statistics collection has been enabled.

Queue Statistics messages are generated only for queues for which statistics data has been collected in the previous time period.

The same queue can have several put operations and get operations through several Object Handles. Some Object Handles might have been opened before statistics collection was enabled, but others were opened afterwards. Therefore, it is possible for the queue statistics to record the activity of some put operations and get operations, and not all.

To ensure that the Queue Statistics are recording the activity of all applications, you must close and reopen new Object Handles on the queue, or queues, that you are monitoring. The best way to achieve this, is to end and restart all applications after enabling statistics collection.

To change the value of the queue attribute STATQ, use the MQSC command, ALTER QLOCAL and specify the parameter STATQ. The queue attribute STATQ can have the following values:

ON

Queue statistics information is collected for every connection to the queue manager that opens the queue.

OFF

Queue statistics information for this queue is not collected.

QMGR

The collection of queue statistics information for this queue is controlled according to the value of the queue manager attribute, STATQ. This is the default value.

To change the value of the queue manager attribute STATQ, use the MQSC command, ALTER QMGR and specify the parameter STATQ. The queue manager attribute STATQ can have the following values:

ON

Queue statistics information is collected for queues that have the queue attribute STATQ set as QMGR

OFF

Queue statistics information is not collected for queues that have the queue attribute STATQ set as QMGR. This is the default value.

NONE

The collection of queue statistics information is disabled for all queues, regardless of the queue attribute STATQ.

If the queue manager attribute STATQ is set to NONE, the collection of queue statistics information is disabled for all queues, regardless of the queue attribute STATQ.

For example, to enable statistics information collection for the queue, Q1, use the following MQSC command:

```
ALTER QLOCAL(Q1) STATQ(ON)
```

To enable statistics information collection for all queues that specify the queue attribute STATQ as QMGR, use the following MQSC command:

```
ALTER QMGR STATQ(ON)
```

distributed *Channel statistics information*

Use the channel attribute STATCHL to control the collection of channel statistics information. You can also set queue manager attributes to control information collection. These attributes are available on distributed platforms and on IBM i.

You can enable or disable channel statistics information collection for individual channels, or for multiple channels. To control individual channels, you must set the channel attribute STATCHL to enable or disable channel statistic information collection. To control many channels together, you enable or disable channel statistics information collection at the queue manager level by using the queue manager attribute STATCHL. For all channels that have the channel attribute STATCHL specified with the value QMGR, channel statistics information collection is controlled at the queue manager level.

Automatically defined cluster-sender channels are not WebSphere MQ objects, so do not have attributes in the same way as channel objects. To control automatically defined cluster-sender channels, use the queue manager attribute STATACLS. This attribute determines whether automatically defined cluster-sender channels within a queue manager are enabled or disabled for channel statistics information collection.

You can set channel statistics information collection to one of the three monitoring levels: low, medium or high. You can set the monitoring level at either object level or at the queue manager level. The choice of which level to use is dependent on your system. Collecting statistics information data might require some instructions that are relatively expensive computationally, so to reduce the impact of channel statistics information collection, the medium and low monitoring options measure a sample of the data at regular intervals rather than collecting data all the time. Table 22 on page 139 summarizes the levels available with channel statistics information collection:

Table 22. Detail level of channel statistics information collection

Level	Description	Usage
Low	Measure a small sample of the data, at regular intervals.	For objects that process a high volume of messages.
Medium	Measure a sample of the data, at regular intervals.	For most objects.
High	Measure all data, at regular intervals.	For objects that process only a few messages per second, on which the most current information is important.

To change the value of the channel attribute STATCHL, use the MQSC command, ALTER CHANNEL and specify the parameter STATCHL.

To change the value of the queue manager attribute STATCHL, use the MQSC command, ALTER QMGR and specify the parameter STATCHL.

To change the value of the queue manager attribute STATACLS, use the MQSC command, ALTER QMGR and specify the parameter STATACLS.

The channel attribute, STATCHL, can have the following values:

LOW
Channel statistics information is collected with a low level of detail.

MEDIUM
Channel statistics information is collected with a medium level of detail.

HIGH

Channel statistics information is collected with a high level of detail.

OFF

Channel statistics information is not collected for this channel.

QMGR

The channel attribute is set as QMGR. The collection of statistics information for this channel is controlled by the value of the queue manager attribute, STATCHL.

This is the default value.

The queue manager attribute, STATCHL, can have the following values:

LOW

Channel statistics information is collected with a low level of detail, for all channels that have the channel attribute STATCHL set as QMGR.

MEDIUM

Channel statistics information is collected with a medium level of detail, for all channels that have the channel attribute STATCHL set as QMGR.

HIGH

Channel statistics information is collected with a high level of detail, for all channels that have the channel attribute STATCHL set as QMGR.

OFF

Channel statistics information is not collected for all channels that have the channel attribute STATCHL set as QMGR.

This is the default value.

NONE

The collection of channel statistics information is disabled for all channel, regardless of the channel attribute STATCHL.

The queue manager attribute, STATACLS, can have the following values:

LOW

Statistics information is collected with a low level of detail for automatically defined cluster-sender channels.

MEDIUM

Statistics information is collected with a medium level of detail for automatically defined cluster-sender channels.

HIGH

Statistics information is collected with a high level of detail for automatically defined cluster-sender channels.

OFF

Statistics information is not for automatically defined cluster-sender channels.

QMGR

The collection of statistics information for automatically defined cluster-sender channels is controlled by the value of the queue manager attribute, STATCHL.

This is the default value.

For example, to enable statistics information collection, with a medium level of detail, for the sender channel QM1.TO.QM2, use the following MQSC command:

```
ALTER CHANNEL(QM1.TO.QM2) CHLTYPE(SDR) STATCHL(MEDIUM)
```

To enable statistics information collection, at a medium level of detail, for all channels that specify the channel attribute STATCHL as QMGR, use the following MQSC command:

```
ALTER QMGR STATCHL(MEDIUM)
```

To enable statistics information collection, at a medium level of detail, for all automatically defined cluster-sender channels, use the following MQSC command:

```
ALTER QMGR STATACLS(MEDIUM)
```

Statistics message generation

Statistics messages are generated at configured intervals, and when a queue manager shuts down in a controlled fashion.

The configured interval is controlled by the STATINT queue manager attribute, which specifies the interval, in seconds, between the generation of statistics messages. The default statistics interval is 1800 seconds (30 minutes). To change the statistics interval, use the MQSC command ALTER QMGR and specify the STATINT parameter. For example, to change the statistics interval to 900 seconds (15 minutes) use the following MQSC command:

```
ALTER QMGR STATINT(900)
```

To write the currently collected statistics data to the statistics queue before the statistics collection interval is due to expire, use the MQSC command RESET QMGR TYPE(STATISTICS). Issuing this command causes the collected statistics data to be written to the statistics queue and a new statistics data collection interval to begin.

Displaying accounting and statistics information

To use the information recorded in accounting and statistics messages, run an application such as the **amqsmon** sample program to transform the recorded information into a suitable format

Accounting and statistics messages are written to the system accounting and statistics queues. **amqsmon** is a sample program supplied with WebSphere MQ that processes messages from the accounting and statistics queues and displays the information to the screen in a readable form.

Because **amqsmon** is a sample program, you can use the supplied source code as template for writing your own application to process accounting or statistics messages, or modify the **amqsmon** source code to meet your own particular requirements.

amqsmon (Display formatted monitoring information)

Use the **amqsmon** sample program to display in a readable format the information contained within accounting and statistics messages. The **amqsmon** program reads accounting messages from the

accounting queue, SYSTEM.ADMIN.ACCOUNTING.QUEUE. and reads statistics messages from the statistics queue, SYSTEM.ADMIN.STATISTICS.QUEUE.

Syntax
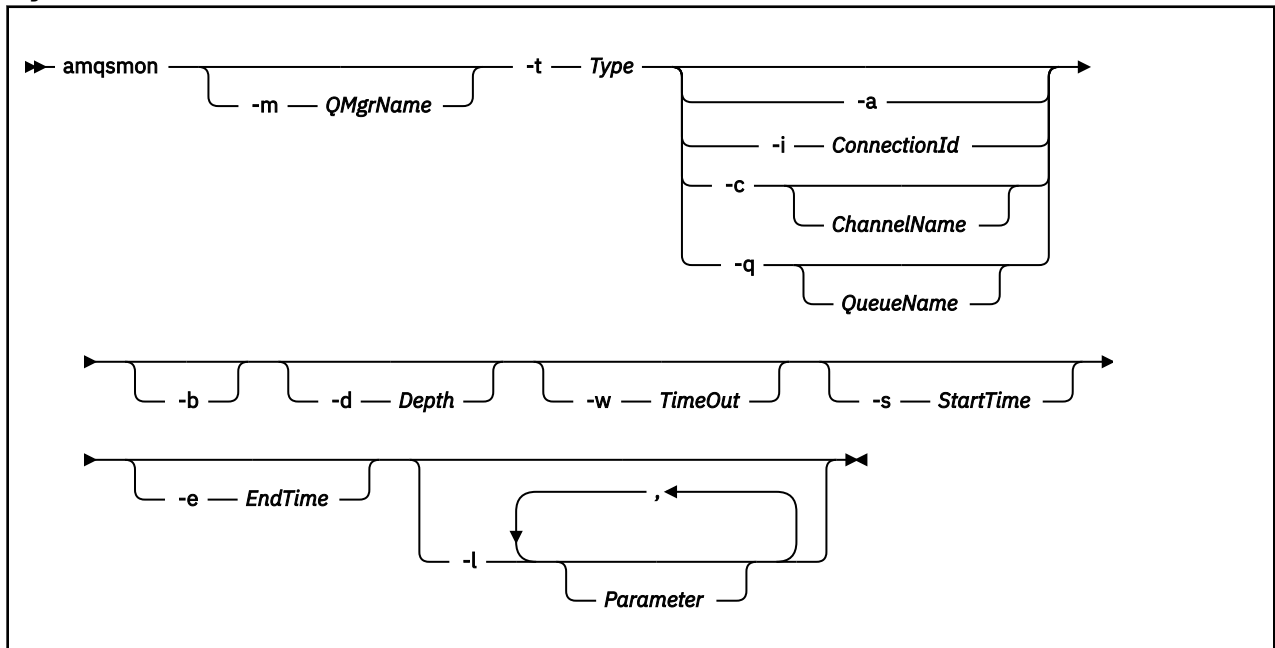
Required parameters

-t *Type*

The type of messages to process. Specify *Type* as one of the following:

accounting

Accounting records are processed. Messages are read from the system queue, SYSTEM.ADMIN.ACCOUNTING.QUEUE.

statistics

Statistics records are processed. Messages are read from the system queue, SYSTEM.ADMIN.STATISTICS.QUEUE.

Optional Parameters

-m *QMgrName*

The name of the queue manager from which accounting or statistics messages are to be processed.

If you do not specify this parameter, the default queue manager is used.

-a

Process messages containing MQI records only.

Only display MQI records. Messages not containing MQI records will always be left on the queue they were read from.

-q *QueueName*

QueueName is an optional parameter.

If *QueueName* is not supplied:	Displays queue accounting and queue statistics records only.

If *QueueName* is supplied:	Displays queue accounting and queue statistics records for the queue specified by *QueueName* only.
	If *-b* is not specified then the accounting and statistics messages from which the records came are discarded. Since accounting and statistics messages can also contain records from other queues, if *-b* is not specified then unseen records can be discarded.

-c *ChannelName*
   ChannelName is an optional parameter.

If *ChannelName* is not supplied:	Displays channel statistics records only.
If *ChannelName* is supplied:	Displays channel statistics records for the channel specified by *ChannelName* only.
	If *-b* is not specified then the statistics messages from which the records came are discarded. Since statistics messages can also contain records from other channels, if *-b* is not specified then unseen records can be discarded.

This parameter is available when displaying statistics messages only, (*-t statistics*).

-i *ConnectionId*
   Displays records related to the connection identifier specified by *ConnectionId* only.

   This parameter is available when displaying accounting messages only, (*-t accounting*).

   If *-b* is not specified then the statistics messages from which the records came are discarded. Since statistics messages can also contain records from other channels, if *-b* is not specified then unseen records can be discarded.

-b
   Browse messages.

   Messages are retrieved non-destructively.

-d *Depth*
   The maximum number of messages that can be processed.

   If you do not specify this parameter, then an unlimited number of messages can be processed.

-w *TimeOut*
   Time maximum number of seconds to wait for a message to become available.

   If you do not specify this parameter, amqsmon will end once there are no more messages to process.

-s *StartTime*
   Process messages put after the specified *StartTime* only.

   StartTime is specified in the format `yyyy-mm-dd  hh.mm.ss`. If a date is specified without a time, then the time will default to `00.00.00` on the date specified. Times are in GMT.

   For the effect of not specifying this parameter, see Note 1.

-e *EndTime*
   Process messages put before the specified *EndTime* only.

   The *EndTime* is specified in the format `yyyy-mm-dd  hh.mm.ss`. If a date is specified without a time, then the time will default to `23.59.59` on the date specified. Times are in GMT.

   For the effect of not specifying this parameter, see Note 1.

-l *Parameter*

Only display the selected fields from the records processed. *Parameter* is a comma-separated list of integer values, with each integer value mapping to the numeric constant of a field, see amqsmon example 5.

If you do not specify this parameter, then all available fields are displayed.

Note:

1. If you do not specify *-s StartTime* or *-e EndTime,* the messages that can be processed are not restricted by put time.

amqsmon examples

Use this page to view examples of running the amqsmon (Display formatted monitoring information) sample program

1. The following command displays all MQI statistics messages from queue manager `saturn.queue.manager`:

```
amqsmon -m saturn.queue.manager -t statistics -a
```

The output from this command follows:

```
RecordType: MQIStatistics
QueueManager: 'saturn.queue.manager'
IntervalStartDate: '2005-04-30'
IntervalStartTime: '15.09.02'
IntervalEndDate: '2005-04-30'
IntervalEndTime: '15.39.02'
CommandLevel: 600
ConnCount: 23
ConnFailCount: 0
ConnsMax: 8
DiscCount: [17, 0, 0]
OpenCount: [0, 80, 1, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0]
OpenFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
CloseCount: [0, 73, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
CloseFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
InqCount: [4, 2102, 0, 0, 0, 46, 0, 0, 0, 0, 0, 0, 0]
InqFailCount: [0, 31, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
SetCount: [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
SetFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
PutCount: [26, 1]
PutFailCount: 0
Put1Count: [40, 0]
Put1FailCount: 0
PutBytes: [57064, 12320]
GetCount: [18, 1]
GetBytes: [52, 12320]
GetFailCount: 2254
BrowseCount: [18, 60]
BrowseBytes: [23784, 30760]
BrowseFailCount: 9
CommitCount: 0
CommitFailCount: 0
BackCount: 0
ExpiredMsgCount: 0
PurgeCount: 0
```

2. The following command displays all queue statistics messages for queue LOCALQ on queue manager `saturn.queue.manager`:

```
amqsmon -m saturn.queue.manager -t statistics -q LOCALQ
```

The output from this command follows:

```
RecordType: QueueStatistics
QueueManager: 'saturn.queue.manager'
```

```
IntervalStartDate: '2005-04-30'
IntervalStartTime: '15.09.02'
IntervalEndDate: '2005-04-30'
IntervalEndTime: '15.39.02'
CommandLevel: 600
ObjectCount: 3
QueueStatistics:
  QueueName: 'LOCALQ'
  CreateDate: '2005-03-08'
  CreateTime: '17.07.02'
  QueueType: Predefined
  QueueDefinitionType: Local
  QMinDepth: 0
  QMaxDepth: 18
  AverageQueueTime: [29827281, 0]
  PutCount: [26, 0]
  PutFailCount: 0
  Put1Count: [0, 0]
  Put1FailCount: 0
  PutBytes: [88, 0]
  GetCount: [18, 0]
  GetBytes: [52, 0]
  GetFailCount: 0
  BrowseCount: [0, 0]
  BrowseBytes: [0, 0]
  BrowseFailCount: 1
  NonQueuedMsgCount: 0
  ExpiredMsgCount: 0
  PurgedMsgCount: 0
```

3. The following command displays all of the statistics messages recorded since 15:30 on 30 April 2005 from queue manager saturn.queue.manager.

```
amqsmon -m saturn.queue.manager -t statistics -s "2005-04-30 15.30.00"
```

The output from this command follows:

```
RecordType: MQIStatistics
QueueManager: 'saturn.queue.manager'
IntervalStartDate: '2005-04-30'
IntervalStartTime: '15.09.02'
IntervalEndDate: '2005-04-30'
IntervalEndTime: '15.39.02'
CommandLevel: 600
ConnCount: 23
ConnFailCount: 0
ConnsMax: 8
DiscCount: [17, 0, 0]
OpenCount: [0, 80, 1, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0]
   ...
RecordType: QueueStatistics
QueueManager: 'saturn.queue.manager'
IntervalStartDate: '2005-04-30'
IntervalStartTime: '15.09.02'
IntervalEndDate: '2005-04-30'
IntervalEndTime: '15.39.02'
CommandLevel: 600
ObjectCount: 3
QueueStatistics: 0
  QueueName: 'LOCALQ'
  CreateDate: '2005-03-08'
  CreateTime: '17.07.02'
  QueueType: Predefined
     ...
QueueStatistics: 1
  QueueName: 'SAMPLEQ'
  CreateDate: '2005-03-08'
  CreateTime: '17.07.02'
  QueueType: Predefined
     ...
```

4. The following command displays all accounting messages recorded on 30 April 2005 from queue manager `saturn.queue.manager`:

```
amqsmon -m saturn.queue.manager -t accounting -s "2005-04-30" -e "2005-04-30"
```

The output from this command follows:

```
RecordType: MQIAccounting
QueueManager: 'saturn.queue.manager'
IntervalStartDate: '2005-04-30'
IntervalStartTime: '15.09.29'
IntervalEndDate: '2005-04-30'
IntervalEndTime: '15.09.30'
CommandLevel: 600
ConnectionId: x'414d514354525245563120202020202020208d0b3742010a0020'
SeqNumber: 0
ApplicationName: 'amqsput'
ApplicationPid: 8572
ApplicationTid: 1
UserId: 'admin'
ConnDate: '2005-03-16'
ConnTime: '15.09.29'
DiscDate: '2005-03-16'
DiscTime: '15.09.30'
DiscType: Normal
OpenCount: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
OpenFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
CloseCount: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
CloseFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
PutCount: [1, 0]
PutFailCount: 0
PutBytes: [4, 0]
GetCount: [0, 0]
GetFailCount: 0
GetBytes: [0, 0]
BrowseCount: [0, 0]
BrowseFailCount: 0
BrowseBytes: [0, 0]
CommitCount: 0
CommitFailCount: 0
BackCount: 0
InqCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
InqFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
SetCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
SetFailCount: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

RecordType: MQIAccounting
QueueManager: 'saturn.queue.manager'
IntervalStartDate: '2005-03-16'
IntervalStartTime: '15.16.22'
IntervalEndDate: '2005-03-16'
IntervalEndTime: '15.16.22'
CommandLevel: 600
ConnectionId: x'414d514354525245563120202020202020208d0b3742010c0020'
SeqNumber: 0
ApplicationName: 'runmqsc'
ApplicationPid: 8615
ApplicationTid: 1
    ...
```

5. The following command browses the accounting queue and displays the application name and connection identifier of every application for which MQI accounting information is available:

```
amqsmon -m saturn.queue.manager -t accounting -b -a -l 7006,3024
```

The output from this command follows:

```
ConnectionId: x'414d514354525245563120202020202020208d0b374203090020'
ApplicationName: 'runmqsc'

ConnectionId: x'414d514354525245563120202020202020208d0b3742010a0020'
ApplicationName: 'amqsput'
```

```
ConnectionId: x'414d5143545245563120202020202020208d0b3742010c0020'
ApplicationName: 'runmqsc'

ConnectionId: x'414d5143545245563120202020202020208d0b3742010d0020'
ApplicationName: 'amqsput'

ConnectionId: x'414d5143545245563120202020202020208d0b3742150d0020'
ApplicationName: 'amqsget'

5 Records Processed.
```

Accounting and statistics message reference

Use this page to obtain an overview of the format of accounting and statistics messages and the information returned in these messages

Accounting and statistics message messages are standard WebSphere MQ messages containing a message descriptor and message data. The message data contains information about the MQI operations performed by WebSphere MQ applications, or information about the activities occurring in a WebSphere MQ system.

Message descriptor

- An MQMD structure

Message data

- A PCF header (MQCFH)
- Accounting or statistics message data that is always returned
- Accounting or statistics message data that is returned if available

Accounting and statistics message format

Use this page as an example of the structure of an MQI accounting message

Table 23. MQI accounting message structure

MQMD structure	Accounting message header MQCFH structure	MQI accounting message data [1]
Structure identifier Structure version Report options Message type Expiration time Feedback code Encoding Coded character set ID Message format Message priority Persistence Message identifier Correlation identifier Backout count Reply-to queue Reply-to queue manager User identifier Accounting token Application identity data Application type Application name Put date Put time Application origin data Group identifier Message sequence number Offset Message flags Original length	Structure type Structure length Structure version Command identifier Message sequence number Control options Completion code Reason code Parameter count	Queue manager Interval start date Interval start time Interval end date Interval end time Command level Connection identifier Sequence number Application name Application process identifier Application thread identifier User identifier Connection date Connection time Connection name Channel name Disconnect date Disconnect time Disconnect type Open count Open fail count Close count Close fail count Put count Put fail count Put1 count Put1 fail count Put bytes Get count Get fail count Get bytes Browse count Browse fail count Browse bytes Commit count Commit fail count Backout count Inquire count Inquire fail count Set count Set fail count

Note:

1. The parameters shown are those returned for an MQI accounting message. The actual accounting or statistics message data depends on the message category.

Accounting and statistics message MQMD (message descriptor)

Use this page to understand the differences between the message descriptor of accounting and statistics messages and the message descriptor of event messages

The parameters and values in the message descriptor of accounting and statistics message are the same as in the message descriptor of event messages, with the following exception:

Format

Description:	Format name of message data.
Data type:	MQCHAR8.
Value:	**MQFMT_ADMIN** Admin message.

Some of the parameters contained in the message descriptor of accounting and statistics message contain fixed data supplied by the queue manager that generated the message.

The MQMD also specifies the name of the queue manager (truncated to 28 characters) that put the message, and the date and time when the message was put on the accounting, or statistics, queue.

Message data in accounting and statistics messages

The message data in accounting and statistics messages is based on the programmable command format (PCF), which is used in PCF command inquiries and responses. The message data in accounting and statistics messages consists of a PCF header (MQCFH) and an accounting or statistics report.

Accounting and statistics message MQCFH (PCF header)

The message header of accounting and statistics messages is an MQCFH structure. The parameters and values in the message header of accounting and statistics message are the same as in the message header of event messages, with the following exceptions:

Command

Description:	Command identifier. This identifies the accounting or statistics message category.
Data type:	MQLONG.
Values:	**MQCMD_ACCOUNTING_MQI** MQI accounting message. **MQCMD_ACCOUNTING_Q** Queue accounting message. **MQCMD_STATISTICS_MQI** MQI statistics message. **MQCMD_STATISTICS_Q** Queue statistics message. **MQCMD_STATISTICS_CHANNEL** Channel statistics message.

Version

Description:	Structure version number.
Data type:	MQLONG.
Value:	**MQCFH_VERSION_3** Version-3 for accounting and statistics messages.

Accounting and statistics message data

The content of accounting and statistics message data is dependent on the category of the accounting or statistics message, as follows:

MQI accounting message
  MQI accounting message data consists of a number of PCF parameters, but no PCF groups.

Queue accounting message
  Queue accounting message data consists of a number of PCF parameters, and in the range 1 through 100 *QAccountingData* PCF groups.

MQI statistics message
  MQI statistics message data consists of a number of PCF parameters, but no PCF groups.

Queue statistics message
  Queue statistics message data consists of a number of PCF parameters, and in the range 1 through 100 *QStatisticsData* PCF groups.

Channel statistics message
  Channel statistics message data consists of a number of PCF parameters, and in the range 1 through 100 *ChlStatisticsData* PCF groups.

MQI accounting message data

Use this page to view the structure of an MQI accounting message

Message name:	MQI accounting message.
Platforms:	All, except WebSphere MQ for z/OS.
System queue:	SYSTEM.ADMIN.ACCOUNTING.QUEUE.

QueueManager

Description:	The name of the queue manager
Identifier:	MQCA_Q_MGR_NAME
Data type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Returned:	Always

IntervalStartDate

Description:	The date of the start of the monitoring period
Identifier:	MQCAMO_START_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH
Returned:	Always

IntervalStartTime

Description:	The time of the start of the monitoring period
Identifier:	MQCAMO_START_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	Always

IntervalEndDate

Description:	The date of the end of the monitoring period
Identifier:	MQCAMO_END_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH
Returned:	Always

IntervalEndTime

Description:	The time of the end of the monitoring period
Identifier:	MQCAMO_END_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	Always

CommandLevel

Description:	The queue manager command level
Identifier:	MQIA_COMMAND_LEVEL
Data type:	MQCFIN
Returned:	Always

ConnectionId

Description:	The connection identifier for the WebSphere MQ connection
Identifier:	MQBACF_CONNECTION_ID
Data type:	MQCFBS
Maximum length:	MQ_CONNECTION_ID_LENGTH
Returned:	Always

SeqNumber

Description:	The sequence number. This value is incremented for each subsequent record for long running connections.
Identifier:	MQIACF_SEQUENCE_NUMBER
Data type:	MQCFIN
Returned:	Always

ApplicationName

Description:	The name of the application. The contents of this field are equivalent to the contents of the *PutApplName* field in the message descriptor.
Identifier:	MQCACF_APPL_NAME
Data type:	MQCFST
Maximum length:	MQ_APPL_NAME_LENGTH
Returned:	Always

ApplicationPid

Description:	The operating system process identifier of the application
Identifier:	MQIACF_PROCESS_ID
Data type:	MQCFIN
Returned:	Always

ApplicationTid

Description:	The WebSphere MQ thread identifier of the connection in the application
Identifier:	MQIACF_THREAD_ID
Data type:	MQCFIN
Returned:	Always

UserId

Description:	The user identifier context of the application
Identifier:	MQCACF_USER_IDENTIFIER
Data type:	MQCFST
Maximum length:	MQ_USER_ID_LENGTH
Returned:	Always

ConnDate

Description:	Date of MQCONN operation
Identifier:	MQCAMO_CONN_DATE
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	When available

ConnTime

Description:	Time of MQCONN operation
Identifier:	MQCAMO_CONN_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	When available

ConnName

Description:	Connection name for client connection
Identifier:	MQCACH_CONNECTION_NAME
Data type:	MQCFST
Maximum length:	MQ_CONN_NAME_LENGTH
Returned:	When available

ChannelName

    Description: Channel name for client connection

    Identifier: MQCACH_CHANNEL_NAME

    Data type: MQCFST

    Maximum length: MQ_CHANNEL_NAME_LENGTH

    Returned: When available

DiscDate

    Description: Date of MQDISC operation

    Identifier: MQCAMO_DISC_DATE

    Data type: MQCFST

    Maximum length: MQ_DATE_LENGTH

    Returned: When available

DiscTime

    Description: Time of MQDISC operation

    Identifier: MQCAMO_DISC_TIME

    Data type: MQCFST

    Maximum length: MQ_TIME_LENGTH

    Returned: When available

DiscType

    Description: Type of disconnect

    Identifier: MQIAMO_DISC_TYPE

    Data type: MQCFIN

    Values: The possible values are:

                          MQDISCONNECT_NORMAL
                            Requested by application

                          MQDISCONNECT_IMPLICIT
                            Abnormal application termination

                          MQDISCONNECT_Q_MGR
                            Connection broken by queue manager

    Returned: When available

OpenCount

    Description: The number of objects opened. This parameter is an integer list indexed by object type, see Reference note 1.

    Identifier: MQIAMO_OPENS

    Data type: MQCFIL

    Returned: When available

OpenFailCount

Description:	The number of unsuccessful attempts to open an object. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_OPENS_FAILED
Data type:	MQCFIL
Returned:	When available

CloseCount

Description:	The number of objects closed. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_CLOSES
Data type:	MQCFIL
Returned:	When available

CloseFailCount

Description:	The number of unsuccessful attempts to close an object. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_CLOSES_FAILED
Data type:	MQCFIL
Returned:	When available

PutCount

Description:	The number persistent and nonpersistent messages successfully put to a queue, with the exception of messages put using the MQPUT1 call. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_PUTS
Data type:	MQCFIL
Returned:	When available

PutFailCount

Description:	The number of unsuccessful attempts to put a message
Identifier:	MQIAMO_PUTS_FAILED
Data type:	MQCFIN
Returned:	When available

Put1Count

Description:	The number of persistent and nonpersistent messages successfully put to the queue using MQPUT1 calls. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_PUT1S
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*

Returned:	When available

Put1FailCount

Description:	The number of unsuccessful attempts to put a message using MQPUT1 calls
Identifier:	MQIAMO_PUT1S_FAILED
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Returned:	When available

PutBytes

Description:	The number bytes written using put calls for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_PUT_BYTES
Data type:	MQCFIL64
Returned:	When available

GetCount

Description:	The number of successful destructive MQGET calls for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_GETS
Data type:	MQCFIL
Returned:	When available

GetFailCount

Description:	The number of failed destructive MQGET calls
Identifier:	MQIAMO_GETS_FAILED
Data type:	MQCFIN
Returned:	When available

GetBytes

Description:	Total number of bytes retrieved for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_GET_BYTES
Data type:	MQCFIL64
Returned:	When available

BrowseCount

Description:	The number of successful non-destructive MQGET calls for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_BROWSES

Data type:	MQCFIL
Returned:	When available

BrowseFailCount

Description:	The number of unsuccessful non-destructive MQGET calls
Identifier:	MQIAMO_BROWSES_FAILED
Data type:	MQCFIN
Returned:	When available

BrowseBytes

Description:	Total number of bytes browsed for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_BROWSE_BYTES
Data type:	MQCFIL64
Returned:	When available

CommitCount

Description:	The number of successful transactions. This number includes those transactions committed implicitly by the connected application. Commit requests where there is no outstanding work are included in this count.
Identifier:	MQIAMO_COMMITS
Data type:	MQCFIN
Returned:	When available

CommitFailCount

Description:	The number of unsuccessful attempts to complete a transaction
Identifier:	MQIAMO_COMMITS_FAILED
Data type:	MQCFIN
Returned:	When available

BackCount

Description:	The number of backouts processed, including implicit backouts due to abnormal disconnection
Identifier:	MQIAMO_BACKOUTS
Data type:	MQCFIN
Returned:	When available

InqCount

Description:	The number of successful objects inquired upon. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_INQS
Data type:	MQCFIL
Returned:	When available

InqFailCount

Description:	The number of unsuccessful object inquire attempts. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_INQS_FAILED
Data type:	MQCFIL
Returned:	When available

SetCount

Description:	The number of successful MQSET calls. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_SETS
Data type:	MQCFIL
Returned:	When available

SetFailCount

Description:	The number of unsuccessful MQSET calls. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_SETS_FAILED
Data type:	MQCFIL
Returned:	When available

SubCountDur

Description:	The number of succesful subscribe requests which created, altered or resumed durable subscriptions. This is an array of values indexed by the type of operation

0 = The number of subscriptions created

1 = The number of subscriptions altered

2 = The number of subscriptions resumed |
Identifier:	MQIAMO_SUBS_DUR
Data type:	MQCFIL
Returned:	When available.

SubCountNDur

Description:	The number of succesful subscribe requests which created, altered or resumed non-durable subscriptions. This is an array of values indexed by the type of operation

0 = The number of subscriptions created

1 = The number of subscriptions altered

2 = The number of subscriptions resumed |
Identifier:	MQIAMO_SUBS_NDUR
Data type:	MQCFIL
Returned:	When available.

SubFailCount

Description:	The number of unsuccessful Subscribe requests.
Identifier:	MQIAMO_SUBS_FAILED
Data type:	MQCFIN
Returned:	When available.

UnsubCountDur

Description:	The number of succesful unsubscribe requests for durable subscriptions. This is an array of values indexed by the type of operation
	0 - The subscription was closed but not removed
	1 - The subscription was closed and removed
Identifier:	MQIAMO_UNSUBS_DUR
Data type:	MQCFIL
Returned:	When available.

UnsubCountNDur

Description:	The number of succesful unsubscribe requests for durable subscriptions. This is an array of values indexed by the type of operation
	0 - The subscription was closed but not removed
	1 - The subscription was closed and removed
Identifier:	MQIAMO_UNSUBS_NDUR
Data type:	MQCFIL
Returned:	When available.

UnsubFailCount

Description:	The number of unsuccessful unsubscribe requests.
Identifier:	MQIAMO_UNSUBS_FAILED
Data type:	MQCFIN
Returned:	When available.

SubRqCount

Description:	The number of successful MQSUBRQ requests.
Identifier:	MQIAMO_SUBRQS
Data type:	MQCFIN
Returned:	When available.

SubRqFailCount

Description:	The number of unsuccessful MQSUB requests.
Identifier:	MQIAMO_SUBRQS_FAILED
Data type:	MQCFIN
Returned:	When available.

CBCount

Description:	The number of successful MQCB requests. This is an array of values indexed by the type of operation
	0 - A callback was created or altered
	1 - A callback was removed
	2 - A callback was resumed
	3 - A callback was suspended
Identifier:	MQIAMO_CBS
Data type:	MQCFIN
Returned:	When available.

CBFailCount

Description:	The number of unsuccessful MQCB requests.
Identifier:	MQIAMO_CBS_FAILED
Data type:	MQCFIN
Returned:	When available.

CtlCount

Description:	The number of successful MQCTL requests. This is an array of values indexed by the type of operation
	0 - The connection was started
	1 - The connection was stopped
	2 - The connection was resumed
	3 - The connection was suspended
Identifier:	MQIAMO_CTLS
Data type:	MQCFIL
Returned:	When available.

CtlFailCount

Description:	The number of unsuccessful MQCTL requests.
Identifier:	MQIAMO_CTLS_FAILED
Data type:	MQCFIN
Returned:	When available.

StatCount

Description:	The number of successful MQSTAT requests.
Identifier:	MQIAMO_STATS.
Data type:	MQCFIN
Returned:	When available.

StatFailCount

Description:	The number of unsuccessful MQSTAT requests.
Identifier:	MQIAMO_STATS_FAILED
Data type:	MQCFIN
Returned:	When available.

PutTopicCount

Description:	The number persistent and nonpersistent messages successfully put to a topic, with the exception of messages put using the MQPUT1 call. This parameter is an integer list indexed by persistence value, see Reference note 2.
	Note: Messages put using a queue alias which resolve to a topic are included in this value.
Identifier:	MQIAMO_TOPIC_PUTS
Data type:	MQCFIL
Returned:	When available.

PutTopicFailCount

Description:	The number of unsuccessful attempts to put a message to a topic.
Identifier:	MQIAMO_TOPIC_PUTS_FAILED
Data type:	MQCFIN
Returned:	When available.

Put1TopicCount

Description:	The number of persistent and nonpersistent messages successfully put to a topic using MQPUT1 calls. This parameter is an integer list indexed by persistence value, see Reference note 2.
	Note: Messages put using a queue alias which resolve to a topic are included in this value.
Identifier:	MQIAMO_TOPIC_PUT1S
Data type:	MQCFIL
Returned:	When available.

Put1TopicFailCount

Description:	The number of unsuccessful attempts to put a message to a topic using MQPUT1 calls.
Identifier:	MQIAMO_TOPIC_PUT1S_FAILED
Data type:	MQCFIN
Returned:	When available.

PutTopicBytes

Description: The number bytes written using put calls for persistent and nonpersistent messages which resolve to a publish operation. This is number of bytes put by the application and not the resultant number of bytes delivered to subscribers. This parameter is an integer list indexed by persistence value, see Reference note 2.

Identifier: MQIAMO64_TOPIC_PUT_BYTES

Data type: MQCFIL64

Returned: When available.

Queue accounting message data

Use this page to view the structure of a queue accounting message

Message name:	Queue accounting message.
Platforms:	All, except WebSphere MQ for z/OS.
System queue:	SYSTEM.ADMIN.ACCOUNTING.QUEUE.

QueueManager

Description: The name of the queue manager

Identifier: MQCA_Q_MGR_NAME

Data type: MQCFST

Maximum length: MQ_Q_MGR_NAME_LENGTH

Returned: Always

IntervalStartDate

Description: The date of the start of the monitoring period

Identifier: MQCAMO_START_DATE

Data type: MQCFST

Maximum length: MQ_DATE_LENGTH

Returned: Always

IntervalStartTime

Description: The time of the start of the monitoring period

Identifier: MQCAMO_START_TIME

Data type: MQCFST

Maximum length: MQ_TIME_LENGTH

Returned: Always

IntervalEndDate

Description: The date of the end of the monitoring period

Identifier: MQCAMO_END_DATE

Data type: MQCFST

Maximum length: MQ_DATE_LENGTH

Returned: Always

IntervalEndTime

Description:	The time of the end of the monitoring period
Identifier:	MQCAMO_END_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	Always

CommandLevel

Description:	The queue manager command level
Identifier:	MQIA_COMMAND_LEVEL
Data type:	MQCFIN
Returned:	Always

ConnectionId

Description:	The connection identifier for the WebSphere MQ connection
Identifier:	MQBACF_CONNECTION_ID
Data type:	MQCFBS
Maximum length:	MQ_CONNECTION_ID_LENGTH
Returned:	Always

SeqNumber

Description:	The sequence number. This value is incremented for each subsequent record for long running connections.
Identifier:	MQIACF_SEQUENCE_NUMBER
Data type:	MQCFIN
Returned:	Always

ApplicationName

Description:	The name of the application. The contents of this field are equivalent to the contents of the PutApplName field in the message descriptor.
Identifier:	MQCACF_APPL_NAME
Data type:	MQCFST
Maximum length:	MQ_APPL_NAME_LENGTH
Returned:	Always

ApplicationPid

Description:	The operating system process identifier of the application
Identifier:	MQIACF_PROCESS_ID
Data type:	MQCFIN
Returned:	Always

ApplicationTid

Description:	The WebSphere MQ thread identifier of the connection in the application
Identifier:	MQIACF_THREAD_ID
Data type:	MQCFIN
Returned:	Always

UserId

Description:	The user identifier context of the application
Identifier:	MQCACF_USER_IDENTIFIER
Data type:	MQCFST
Maximum length:	MQ_USER_ID_LENGTH
Returned:	Always

ObjectCount

Description:	The number of queues accessed in the interval for which accounting data has been recorded. This value is set to the number of *QAccountingData* PCF groups contained in the message.
Identifier:	MQIAMO_OBJECT_COUNT
Data type:	MQCFIN
Returned:	Always

QAccountingData

Description:	Grouped parameters specifying accounting details for a queue
Identifier:	MQGACF_Q_ACCOUNTING_DATA
Data type:	MQCFGR

Parameters in group:	*QName*
	CreateDate
	CreateTime
	QType
	QDefinitionType
	OpenCount
	OpenDate
	OpenTime
	CloseDate
	CloseTime
	PutCount
	PutFailCount
	Put1Count
	Put1FailCount
	PutBytes
	PutMinBytes
	PutMaxBytes
	GetCount
	GetFailCount
	GetBytes
	GetMinBytes
	GetMaxBytes
	BrowseCount
	BrowseFailCount
	BrowseBytes
	BrowseMinBytes
	BrowseMaxBytes
	TimeOnQMin
	TimeOnQAvg
	TimeOnQMax
Returned:	Always

QName

Description:	The name of the queue
Identifier:	MQCA_Q_NAME
Data type:	MQCFST
Included in PCF group:	*QAccountingData*
Maximum length:	MQ_Q_NAME_LENGTH
Returned:	When available

CreateDate

Description:	The date the queue was created
Identifier:	MQCA_CREATION_DATE
Data type:	MQCFST
Included in PCF group:	*QAccountingData*

Maximum length:	MQ_DATE_LENGTH
Returned:	When available

CreateTime

Description:	The time the queue was created
Identifier:	MQCA_CREATION_TIME
Data type:	MQCFST
Included in PCF group:	*QAccountingData*
Maximum length:	MQ_TIME_LENGTH
Returned:	When available

QType

Description:	The type of the queue
Identifier:	MQIA_Q_TYPE
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Value:	MQQT_LOCAL
Returned:	When available

QDefinitionType

Description:	The queue definition type
Identifier:	MQIA_DEFINITION_TYPE
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Values:	Possible values are:
	MQQDT_PREDEFINED
	MQQDT_PERMANENT_DYNAMIC
	MQQDT_TEMPORARY_DYNAMIC
Returned:	When available

OpenCount

Description:	The number of times this queue was opened by the application in this interval
Identifier:	MQIAMO_OPENS
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

OpenDate

Description:	The date the queue was first opened in this recording interval. If the queue was already open at the start of this interval, this value reflects the date the queue was originally opened.
Identifier:	MQCAMO_OPEN_DATE
Data type:	MQCFST
Included in PCF group:	*QAccountingData*
Returned:	When available

OpenTime

Description:	The time the queue was first opened in this recording interval. If the queue was already open at the start of this interval, this value reflects the time the queue was originally opened.
Identifier:	MQCAMO_OPEN_TIME
Data type:	MQCFST
Included in PCF group:	*QAccountingData*
Returned:	When available

CloseDate

Description:	The date of the final close of the queue in this recording interval. If the queue is still open then the value is not returned.
Identifier:	MQCAMO_CLOSE_DATE
Data type:	MQCFST
Included in PCF group:	*QAccountingData*
Returned:	When available

CloseTime

Description:	The time of final close of the queue in this recording interval. If the queue is still open then the value is not returned.
Identifier:	MQCAMO_CLOSE_TIME
Data type:	MQCFST
Included in PCF group:	*QAccountingData*
Returned:	When available

PutCount

Description:	The number of persistent and nonpersistent messages successfully put to the queue, with the exception of MQPUT1 calls. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_PUTS
Data type:	MQCFIL

Included in PCF group:	*QAccountingData*
Returned:	When available

PutFailCount

Description:	The number of unsuccessful attempts to put a message, with the exception of MQPUT1 calls
Identifier:	MQIAMO_PUTS_FAILED
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Returned:	When available

Put1Count

Description:	The number of persistent and nonpersistent messages successfully put to the queue using MQPUT1 calls. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_PUT1S
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

Put1FailCount

Description:	The number of unsuccessful attempts to put a message using MQPUT1 calls
Identifier:	MQIAMO_PUT1S_FAILED
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Returned:	When available

PutBytes

Description:	The total number of bytes put for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_PUT_BYTES
Data type:	MQCFIL64
Included in PCF group:	*QAccountingData*
Returned:	When available

PutMinBytes

Description:	The smallest persistent and nonpersistent message size placed on the queue. This parameter is an integer list indexed by persistence value, see Reference note 2.

Identifier:	MQIAMO_PUT_MIN_BYTES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

PutMaxBytes

Description:	The largest persistent and nonpersistent message size placed on the queue. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_PUT_MAX_BYTES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

GeneratedMsgCount

Description:	The number of generated messages. Generated messages are
	• Queue Depth Hi Events
	• Queue Depth Low Events
Identifier:	MQIAMO_GENERATED_MSGS
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Returned:	When available

GetCount

Description:	The number of successful destructive MQGET calls for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_GETS
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

GetFailCount

Description:	The number of failed destructive MQGET calls
Identifier:	MQIAMO_GETS_FAILED
Data type:	MQCFIN
Included in PCF group:	*QAccountingData*
Returned:	When available

GetBytes

Description:	The number of bytes read in destructive MQGET calls for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_GET_BYTES
Data type:	MQCFIL64
Included in PCF group:	*QAccountingData*
Returned:	When available

GetMinBytes

Description:	The size of the smallest persistent and nonpersistent message retrieved rom the queue. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_GET_MIN_BYTES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

GetMaxBytes

Description:	The size of the largest persistent and nonpersistent message retrieved rom the queue. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_GET_MAX_BYTES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

BrowseCount

Description:	The number of successful non-destructive MQGET calls for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_BROWSES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

BrowseFailCount

Description:	The number of unsuccessful non-destructive MQGET calls
Identifier:	MQIAMO_BROWSES_FAILED
Data type:	MQCFIN

Included in PCF group:	*QAccountingData*
Returned:	When available

BrowseBytes

Description:	The number of bytes read in non-destructive MQGET calls that returned persistent messages
Identifier:	MQIAMO64_BROWSE_BYTES
Data type:	MQCFIL64
Included in PCF group:	*QAccountingData*
Returned:	When available

BrowseMinBytes

Description:	The size of the smallest persistent and nonpersistent message browsed from the queue. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_BROWSE_MIN_BYTES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

BrowseMaxBytes

Description:	The size of the largest persistent and nonpersistent message browsed from the queue. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO_BROWSE_MAX_BYTES
Data type:	MQCFIL
Included in PCF group:	*QAccountingData*
Returned:	When available

CBCount

Description:	The number of successful MQCB requests. This is an array of values indexed by the type of operation
	0 - A callback was created or altered
	1 - A callback was removed
	2 - A callback was resumed
	3 - A callback was suspended
Identifier:	MQIAMO_CBS
Data type:	MQCFIN
Returned:	When available.

CBFailCount

Description:	The number of unsuccessful MQCB requests.
Identifier:	MQIAMO_CBS_FAILED
Data type:	MQCFIN
Returned:	When available.

TimeOnQMin

Description:	The shortest time a persistent and nonpersistent message remained on the queue before being destructively retrieved, in microseconds. For messages retrieved under syncpoint this value does not included the time before the get operation is committed. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_Q_TIME_MIN
Data type:	MQCFIL64
Included in PCF group:	*QAccountingData*
Returned:	When available

TimeOnQAvg

Description:	The average time a persistent and nonpersistent message remained on the queue before being destructively retrieved, in microseconds. For messages retrieved under syncpoint this value does not included the time before the get operation is committed. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_Q_TIME_AVG
Data type:	MQCFIL64
Included in PCF group:	*QAccountingData*
Returned:	When available

TimeOnQMax

Description:	The longest time a persistent and nonpersistent message remained on the queue before being destructively retrieved, in microseconds. For messages retrieved under syncpoint this value does not included the time before the get operation is committed. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_Q_TIME_MAX
Data type:	MQCFIL64
Included in PCF group:	*QAccountingData*
Returned:	When available

MQI statistics message data

Use this page to view the structure of an MQI statistics message

Message name:	MQI statistics message.

Platforms:	All, except WebSphere MQ for z/OS.
System queue:	SYSTEM.ADMIN.STATISTICS.QUEUE.

QueueManager

Description:	Name of the queue manager.
Identifier:	MQCA_Q_MGR_NAME.
Data type:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

IntervalStartDate

Description:	The date at the start of the monitoring period.
Identifier:	MQCAMO_START_DATE.
Data type:	MQCFST.
Maximum length:	MQ_DATE_LENGTH
Returned:	Always.

IntervalStartTime

Description:	The time at the start of the monitoring period.
Identifier:	MQCAMO_START_TIME.
Data type:	MQCFST.
Maximum length:	MQ_TIME_LENGTH
Returned:	Always.

IntervalEndDate

Description:	The date at the end of the monitoring period.
Identifier:	MQCAMO_END_DATE.
Data type:	MQCFST.
Maximum length:	MQ_DATE_LENGTH
Returned:	Always.

IntervalEndTime

Description:	The time at the end of the monitoring period.
Identifier:	MQCAMO_END_TIME.
Data type:	MQCFST.
Maximum length:	MQ_TIME_LENGTH
Returned:	Always.

CommandLevel

Description:	The queue manager command level.
Identifier:	MQIA_COMMAND_LEVEL.

Data type:	MQCFIN.
Returned:	Always.

ConnCount

Description:	The number of successful connections to the queue manager.
Identifier:	MQIAMO_CONNS.
Data type:	MQCFIN.
Returned:	When available.

ConnFailCount

Description:	The number of unsuccessful connection attempts.
Identifier:	MQIAMO_CONNS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

ConnsMax

Description:	The maximum number of concurrent connections in the recording interval.
Identifier:	MQIAMO_CONNS_MAX.
Data type:	MQCFIN.
Returned:	When available.

DiscCount

Description:	The number of disconnects from the queue manager. This is an integer array, indexed by the following constants: • MQDISCONNECT_NORMAL • MQDISCONNECT_IMPLICIT • MQDISCONNECT_Q_MGR
Identifier:	MQIAMO_DISCS.
Data type:	MQCFIL.
Returned:	When available.

OpenCount

Description:	The number of objects successfully opened. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_OPENS.
Data type:	MQCFIL.
Returned:	When available.

OpenFailCount

Description:	The number of unsuccessful open object attempts. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_OPENS_FAILED.

Data type:	MQCFIL.
Returned:	When available.

CloseCount

Description:	The number of objects successfully closed. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_CLOSES.
Data type:	MQCFIL.
Returned:	When available.

CloseFailCount

Description:	The number of successful close object attempts. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_CLOSES_FAILED.
Data type:	MQCFIL.
Returned:	When available.

InqCount

Description:	The number of objects successfully inquired upon. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_INQS.
Data type:	MQCFIL.
Returned:	When available.

InqFailCount

Description:	The number of unsuccessful object inquire attempts. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_INQS_FAILED.
Data type:	MQCFIL.
Returned:	When available.

SetCount

Description:	The number of objects successfully updated (SET). This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_SETS.
Data type:	MQCFIL.
Returned:	When available.

SetFailCount

Description:	The number of unsuccessful SET attempts. This parameter is an integer list indexed by object type, see Reference note 1.
Identifier:	MQIAMO_SETS_FAILED.
Data type:	MQCFIL.

Returned: When available.

PutCount

Description: The number of persistent and nonpersistent messages successfully put to a queue, with the exception of MQPUT1 requests. This parameter is an integer list indexed by persistence value, see Reference note 2.

Identifier: MQIAMO_PUTS.

Data type: MQCFIL.

Returned: When available.

PutFailCount

Description: The number of unsuccessful put message attempts.

Identifier: MQIAMO_PUTS_FAILED.

Data type: MQCFIN.

Returned: When available.

Put1Count

Description: The number of persistent and nonpersistent messages successfully put to a queue using MQPUT1 requests. This parameter is an integer list indexed by persistence value, see Reference note 2

Identifier: MQIAMO_PUT1S.

Data type: MQCFIL.

Returned: When available.

Put1FailCount

Description: The number of unsuccessful attempts to put a persistent and nonpersistent message to a queue using MQPUT1 requests. This parameter is an integer list indexed by persistence value, see Reference note 2

Identifier: MQIAMO_PUT1S_FAILED.

Data type: MQCFIL.

Returned: When available.

PutBytes

Description: The number bytes for persistent and nonpersistent messages written in using put requests. This parameter is an integer list indexed by persistence value, see Reference note 2

Identifier: MQIAMO64_PUT_BYTES.

Data type: MQCFIL64.

Returned: When available.

GetCount

Description: The number of successful destructive get requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2

Identifier: MQIAMO_GETS.

Data type:	MQCFIL.
Returned:	When available.

GetFailCount

Description:	The number of unsuccessful destructive get requests.
Identifier:	MQIAMO_GETS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

GetBytes

Description:	The number of bytes read in destructive gets requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2
Identifier:	MQIAMO64_GET_BYTES.
Data type:	MQCFIL64.
Returned:	When available.

BrowseCount

Description:	The number of successful non-destructive get requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2
Identifier:	MQIAMO_BROWSES.
Data type:	MQCFIL.
Returned:	When available.

BrowseFailCount

Description:	The number of unsuccessful non-destructive get requests.
Identifier:	MQIAMO_BROWSES_FAILED.
Data type:	MQCFIN.
Returned:	When available.

BrowseBytes

Description:	The number of bytes read in non-destructive get requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value, see Reference note 2
Identifier:	MQIAMO64_BROWSE_BYTES.
Data type:	MQCFIL64.
Returned:	When available.

CommitCount

Description:	The number of transactions successfully completed. This number includes transactions committed implicitly by the application disconnecting, and commit requests where there is no outstanding work.
Identifier:	MQIAMO_COMMITS.

Data type: MQCFIN.

Returned: When available.

CommitFailCount

Description:	The number of unsuccessful attempts to complete a transaction.
Identifier:	MQIAMO_COMMITS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

BackCount

Description:	The number of backouts processed, including implicit backout upon abnormal disconnect.
Identifier:	MQIAMO_BACKOUTS.
Data type:	MQCFIN.
Returned:	When available.

ExpiredMsgCount

Description:	The number of persistent and nonpersistent messages that were discarded because they had expired, before they could be retrieved.
Identifier:	MQIAMO_MSGS_EXPIRED.
Data type:	MQCFIN.
Returned:	When available.

PurgeCount

Description:	The number of times the queue has been cleared.
Identifier:	MQIAMO_MSGS_PURGED.
Data type:	MQCFIN.
Returned:	When available.

SubCountDur

Description:	The number of successful Subscribe requests which created, altered or resumed durable subscriptions. This is an array of values indexed by the type of operation
	0 = The number of subscriptions created
	1 = The number of subscriptions altered
	2 = The number of subscriptions resumed
Identifier:	MQIAMO_SUBS_DUR.
Data type:	MQCFIL
Returned:	When available.

SubCountNDur

Description:	The number of successful Subscribe requests which created, altered or resumed non-durable subscriptions. This is an array of values indexed by the type of operation
	0 = The number of subscriptions created
	1 = The number of subscriptions altered
	2 = The number of subscriptions resumed
Identifier:	MQIAMO_SUBS_NDUR.
Data type:	MQCFIL.
Returned:	When available.

SubFailCount

Description:	The number of unsuccessful Subscribe requests.
Identifier:	MQIAMO_SUBS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

UnsubCountDur

Description:	The number of succesful unsubscribe requests for durable subscriptions. This is an array of values indexed by the type of operation
	0 - The subscription was closed but not removed
	1 - The subscription was closed and removed
Identifier:	MQIAMO_UNSUBS_DUR.
Data type:	MQCFIL.
Returned:	When available.

UnsubCountNDur

Description:	The number of succesful unsubscribe requests for non-durable subscriptions. This is an array of values indexed by the type of operation
	0 - The subscription was closed but not removed
	1 - The subscription was closed and removed
Identifier:	MQIAMO_UNSUBS_NDUR.
Data type:	MQCFIL.
Returned:	When available.

UnsubFailCount

Description:	The number of failed unsubscribe requests.
Identifier:	MQIAMO_UNSUBS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

SubRqCount

Description:	The number of successful MQSUBRQ requests.
Identifier:	MQIAMO_SUBRQS
Data type:	MQCFIN
Returned:	When available.

SubRqFailCount

Description:	The number of unsuccessful MQSUBRQ requests.
Identifier:	MQIAMO_SUBRQS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

CBCount

Description:	The number of successful MQCB requests. This is an array of values indexed by the type of operation
	0 - A callback was created or altered
	1 - A callback was removed
	2 - A callback was resumed
	3 - A callback was suspended
Identifier:	MQIAMO_CBS.
Data type:	MQCFIL.
Returned:	When available.

CBFailCount

Description:	The number of unsuccessful MQCB requests.
Identifier:	MQIAMO_CBS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

CtlCount

Description:	The number of successful MQCTL requests. This is an array of values indexed by the type of operation:
	0 - The connection was started
	1 - The connection was stopped
	2 - The connection was resumed
	3 - The connection was suspended
Identifier:	MQIAMO_CTLS.
Data type:	MQCFIL.
Returned:	When available.

CtlFailCount

Description:	The number of unsuccessful MQCTL requests.
Identifier:	MQIAMO_CTLS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

StatCount

Description:	The number of successful MQSTAT requests.
Identifier:	MQIAMO_STATS.
Data type:	MQCFIN.
Returned:	When available.

StatFailCount

Description:	The number of unsuccessful MQSTAT requests.
Identifier:	MQIAMO_STATS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

SubCountDurHighWater

Description: The high-water mark on the number of durable subscriptions during the time interval. This is an array of values indexed by SUBTYPE

0 - The high-water mark for all durable subscriptions in the system

1 - The high-water mark for durable application subscriptions (MQSUBTYPE_API)

2 - The high-water mark for durable admin subscription (MQSUBTYPE_ADMIN)

3 - The high-water mark for durable proxy subscriptions (MQSUBTYPE_PROXY)

Identifier:	MQIAMO_SUB_DUR_HIGHWATER
Data type:	MQCFIL.
Returned:	When available.

SubCountDurLowWater

Description: The low-water mark on the number of durable subscriptions during the time interval. This is an array of values indexed by SUBTYPE.

0 - The low-water mark for all durable subscriptions in the system

1 - The low-water mark for durable application subscriptions (MQSUBTYPE_API)

2 - The low-water mark for durable admin subscriptions (MQSUBTYPE_ADMIN)

3 - The low-water mark for durable proxy subscriptions (MQSUBTYPE_PROXY)

Identifier:	MQIAMO_SUB_DUR_LOWWATER
Data type:	MQCFIL.
Returned:	When available.

SubCountNDurHighWater

Description:	The high-water mark on the number of non-durable subscriptions during the time interval. This is an array of values indexed by SUBTYPE
	0 - The high-water mark for all non-durable subscriptions in the system
	1 - The high-water mark for non-durable application subscriptions (MQSUBTYPE_API)
	2 - The high-water mark for non-durable admin subscription (MQSUBTYPE_ADMIN)
	3 - The high-water mark for non-durable proxy subscriptions (MQSUBTYPE_PROXY)
Identifier:	MQIAMO_SUB_NDUR_HIGHWATER
Data type:	MQCFIL.
Returned:	When available.

SubCountNDurLowWater

Description:	The low-water mark on the number of non-durable subscriptions during the time interval. This is an array of values indexed by SUBTYPE.
	0 - The low-water mark for all non-durable subscriptions in the system
	1 - The low-water mark for non-durable application subscriptions (MQSUBTYPE_API)
	2 - The low-water mark for non-durable admin subscriptions (MQSUBTYPE_ADMIN)
	3 - The low-water mark for non-durable proxy subscriptions (MQSUBTYPE_PROXY)
Identifier:	MQIAMO_SUB_NDUR_LOWWATER
Data type:	MQCFIL.
Returned:	When available.

PutTopicCount

Description:	The number persistent and nonpersistent messages successfully put to a topic, with the exception of messages put using the MQPUT1 call. This parameter is an integer list indexed by persistence value, see Reference note 2.
	Note: Messages put using a queue alias which resolve to a topic are included in this value.
Identifier:	MQIAMO_TOPIC_PUTS.
Data type:	MQCFIL.
Returned:	When available.

PutTopicFailCount

Description:	The number of unsuccessful attempts to put a message to a topic.
Identifier:	MQIAMO_TOPIC_PUTS_FAILED.
Data type:	MQCFIN.
Returned:	When available.

Put1TopicCount

Description:	The number of persistent and nonpersistent messages successfully put to a topic using MQPUT1 calls. This parameter is an integer list indexed by persistence value, see Reference note 2.
	Note: Messages put using a queue alias which resolve to a topic are included in this value.
Identifier:	MQIAMO_TOPIC_PUT1S.
Data type:	MQCFIL.
Returned:	When available.

Put1TopicFailCount

Description:	The number of unsuccessful attempts to put a message to a topic using MQPUT1 calls.
Identifier:	MQIAMO_TOPIC_PUT1S_FAILED.
Data type:	MQCFIN.
Returned:	When available.

PutTopicBytes

Description:	The number bytes written using put calls for persistent and nonpersistent messages which resolve to a publish operation. This is number of bytes put by the application and not the resultant number of bytes delivered to subscribers, see PublishMsgBytes for this value. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_TOPIC_PUT_BYTES.
Data type:	MQCFIL64.
Returned:	When available.

PublishMsgCount

Description:	The number of messages delivered to subscriptions in the time interval. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_PUBLISH_MSG_COUNT
Data type:	MQCFIL.
Returned:	When available.

PublishMsgBytes

Description:	The number of bytes delivered to subscriptions in the time interval. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_PUBLISH_MSG_BYTES
Data type:	MQCFIL64.
Returned:	When available.

Queue statistics message data

Use this page to view the structure of a queue statistics message

Message name:	Queue statistics message.
Platforms:	All, except WebSphere MQ for z/OS.
System queue:	SYSTEM.ADMIN.STATISTICS.QUEUE.

QueueManager

Description:	Name of the queue manager
Identifier:	MQCA_Q_MGR_NAME
Data type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Returned:	Always

IntervalStartDate

Description:	The date at the start of the monitoring period
Identifier:	MQCAMO_START_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH
Returned:	Always

IntervalStartTime

Description:	The time at the start of the monitoring period
Identifier:	MQCAMO_START_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	Always

IntervalEndDate

Description:	The date at the end of the monitoring period
Identifier:	MQCAMO_END_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH
Returned:	Always

IntervalEndTime

Description:	The time at the end of the monitoring period
Identifier:	MQCAMO_END_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	Always

CommandLevel

Description:	The queue manager command level
Identifier:	MQIA_COMMAND_LEVEL
Data type:	MQCFIN
Returned:	Always

ObjectCount

Description:	The number of queue objects accessed in the interval for which statistics data has been recorded. This value is set to the number of QStatisticsData PCF groups contained in the message.
Identifier:	MQIAMO_OBJECT_COUNT
Data type:	MQCFIN
Returned:	Always

QStatisticsData

Description:	Grouped parameters specifying statistics details for a queue
Identifier:	MQGACF_Q_STATISTICS_DATA
Data type:	MQCFGR
Parameters in group:	*QName* *CreateDate* *CreateTime* *QType* *QDefinitionType* *QMinDepth* *QMaxDepth* *AvgTimeOnQ* *PutCount* *PutFailCount* *Put1Count* *Put1FailCount* *PutBytes* *GetCount* *GetFailCount* *GetBytes* *BrowseCount* *BrowseFailCount* *BrowseBytes* *NonQueuedMsgCount* *ExpiredMsgCount* *PurgeCount*
Returned:	Always

QName

Description:	The name of the queue
Identifier:	MQCA_Q_NAME

Data type:	MQCFST
Maximum length:	MQ_Q_NAME_LENGTH
Returned:	Always

CreateDate

Description:	The date when the queue was created
Identifier:	MQCA_CREATION_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH
Returned:	Always

CreateTime

Description:	The time when the queue was created
Identifier:	MQCA_CREATION_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH
Returned:	Always

QType

Description:	The type of the queue
Identifier:	MQIA_Q_TYPE
Data type:	MQCFIN
Value:	MQOT_LOCAL
Returned:	Always

QDefinitionType

Description:	The queue definition type
Identifier:	MQIA_DEFINITION_TYPE
Data type:	MQCFIN
Values:	Possible values are

- MQQDT_PREDEFINED
- MQQDT_PERMANENT_DYNAMIC
- MQQDT_TEMPORARY_DYNAMIC

Returned:	When available

QMinDepth

Description:	The minimum queue depth during the monitoring period
Identifier:	MQIAMO_Q_MIN_DEPTH
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*

Returned:	When available

QMaxDepth

Description:	The maximum queue depth during the monitoring period
Identifier:	MQIAMO_Q_MAX_DEPTH
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

AvgTimeOnQ

Description:	The average latency, in microseconds, of messages destructively retrieved from the queue during the monitoring period. This parameter is an integer list indexed by persistence value, see Reference note 2.
Identifier:	MQIAMO64_AVG_Q_TIME
Data type:	MQCFIL64
Included in PCF group:	*QStatisticsData*
Returned:	When available

PutCount

Description:	The number of persistent and nonpersistent messages successfully put to the queue, with exception of MQPUT1 requests. This parameter is an integer list indexed by persistence value. See Reference note 2.
Identifier:	MQIAMO_PUTS
Data type:	MQCFIL
Included in PCF group:	*QStatisticsData*
Returned:	When available

PutFailCount

Description:	The number of unsuccessful attempts to put a message to the queue
Identifier:	MQIAMO_PUTS_FAILED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

Put1Count

Description:	The number of persistent and nonpersistent messages successfully put to the queue using MQPUT1 calls. This parameter is an integer list indexed by persistence value. See Reference note 2.
Identifier:	MQIAMO_PUT1S
Data type:	MQCFIL

Included in PCF group:	*QStatisticsData*
Returned:	When available

Put1FailCount

Description:	The number of unsuccessful attempts to put a message using MQPUT1 calls
Identifier:	MQIAMO_PUT1S_FAILED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

PutBytes

Description:	The number of bytes written in put requests to the queue
Identifier:	MQIAMO64_PUT_BYTES
Data type:	MQCFIL64
Included in PCF group:	*QStatisticsData*
Returned:	When available

GetCount

Description:	The number of successful destructive get requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value. See Reference note 2.
Identifier:	MQIAMO_GETS
Data type:	MQCFIL
Included in PCF group:	*QStatisticsData*
Returned:	When available

GetFailCount

Description:	The number of unsuccessful destructive get requests
Identifier:	MQIAMO_GETS_FAILED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

GetBytes

Description:	The number of bytes read in destructive put requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value. See Reference note 2.
Identifier:	MQIAMO64_GET_BYTES
Data type:	MQCFIL64

Included in PCF group:	*QStatisticsData*
Returned:	When available

BrowseCount

Description:	The number of successful non-destructive get requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value. See Reference note 2.
Identifier:	MQIAMO_BROWSES
Data type:	MQCFIL
Included in PCF group:	*QStatisticsData*
Returned:	When available

BrowseFailCount

Description:	The number of unsuccessful non-destructive get requests
Identifier:	MQIAMO_BROWSES_FAILED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

BrowseBytes

Description:	The number of bytes read in non-destructive get requests for persistent and nonpersistent messages. This parameter is an integer list indexed by persistence value. See Reference note 2.
Identifier:	MQIAMO64_BROWSE_BYTES
Data type:	MQCFIL64
Included in PCF group:	*QStatisticsData*
Returned:	When available

NonQueuedMsgCount

Description:	The number of messages that bypassed the queue and were transferred directly to a waiting application.

Bypassing a queue can only occur in certain circumstances. This number represents how many times WebSphere MQ was able to bypass the queue, and not the number of times an application was waiting. |
Identifier:	MQIAMO_MSGS_NOT_QUEUED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

ExpiredMsgCount

Description:	The number of persistent and nonpersistent messages that were discarded because they had expired before they could be retrieved.
Identifier:	MQIAMO_MSGS_EXPIRED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

PurgeCount

Description:	The number of messages purged.
Identifier:	MQIAMO_MSGS_PURGED
Data type:	MQCFIN
Included in PCF group:	*QStatisticsData*
Returned:	When available

CBCount

Description:	The number of successful MQCB requests. This is an array of values indexed by the type of operation
	0 - A callback was created or altered
	1 - A callback was removed
	2 - A callback was resumed
	3 - A callback was suspended
Identifier:	MQIAMO_CBS
Data type:	MQCFIN
Returned:	When available.

CBFailCount

Description:	The number of unsuccessful MQCB requests.
Identifier:	MQIAMO_CBS_FAILED
Data type:	MQCFIN
Returned:	When available.

Channel statistics message data

Use this page to view the structure of a channel statistics message

Message name:	Channel statistics message.
Platforms:	All, except WebSphere MQ for z/OS.
System queue:	SYSTEM.ADMIN.STATISTICS.QUEUE.

QueueManager

Description:	The name of the queue manager.

Identifier:	MQCA_Q_MGR_NAME.
Data type:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

IntervalStartDate

Description:	The date at the start of the monitoring period.
Identifier:	MQCAMO_START_DATE.
Data type:	MQCFST.
Maximum length:	MQ_DATE_LENGTH.
Returned:	Always.

IntervalStartTime

Description:	The time at the start of the monitoring period.
Identifier:	MQCAMO_START_TIME.
Data type:	MQCFST.
Maximum length:	MQ_TIME_LENGTH.
Returned:	Always.

IntervalEndDate

Description:	The date at the end of the monitoring period
Identifier:	MQCAMO_END_DATE.
Data type:	MQCFST.
Maximum length:	MQ_DATE_LENGTH.
Returned:	Always.

IntervalEndTime

Description:	The time at the end of the monitoring period
Identifier:	MQCAMO_END_TIME.
Data type:	MQCFST.
Maximum length:	MQ_TIME_LENGTH
Returned:	Always.

CommandLevel

Description:	The queue manager command level.
Identifier:	MQIA_COMMAND_LEVEL.
Data type:	MQCFIN.
Returned:	Always.

ObjectCount

Description:	The number of Channel objects accessed in the interval for which statistics data has been recorded. This value is set to the number of ChlStatisticsData PCF groups contained in the message.
Identifier:	MQIAMO_OBJECT_COUNT
Data type:	MQCFIN.
Returned:	Always.

ChlStatisticsData

Description:	Grouped parameters specifying statistics details for a channel.
Identifier:	MQGACF_CHL_STATISTICS_DATA.
Data type:	MQCFGR.
Parameters in group:	*ChannelName* *ChannelType* *RemoteQmgr* *ConnectionName* *MsgCount* *TotalBytes* *NetTimeMin* *NetTimeAvg* *NetTimeMax* *ExitTimeMin* *ExitTimeAvg* *ExitTimeMax* *FullBatchCount* *IncmplBatchCount* *AverageBatchSize* *PutRetryCount*
Returned:	Always.

ChannelName

Description:	The name of the channel.
Identifier:	MQCACH_CHANNEL_NAME.
Data type:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

ChannelType

Description:	The channel type.
Identifier:	MQIACH_CHANNEL_TYPE.
Data type:	MQCFIN.

Values:	Possible values are:

MQCHT_SENDER
Sender channel.

MQCHT_SERVER
Server channel.

MQCHT_RECEIVER
Receiver channel.

MQCHT_REQUESTER
Requester channel.

MQCHT_CLUSRCVR
Cluster receiver channel.

MQCHT_CLUSSDR
Cluster sender channel.

Returned:	Always.

RemoteQmgr

Description:	The name of the remote queue manager.
Identifier:	MQCA_REMOTE_Q_MGR_NAME.
Data type:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH
Returned:	When available.

ConnectionName

Description:	Connection name of remote queue manager.
Identifier:	MQCACH_CONNECTION_NAME.
Data type:	MQCFST
Maximum length:	MQ_CONN_NAME_LENGTH
Returned:	When available.

MsgCount

Description:	The number of persistent and nonpersistent messages sent or received.
Identifier:	MQIAMO_MSGS.
Data type:	MQCFIN
Returned:	When available.

TotalBytes

Description:	The number of bytes sent or received for persistent and nonpersistent messages.
Identifier:	MQIAMO64_BYTES.
Data type:	MQCFIN64.
Returned:	When available.

NetTimeMin

Description:	The shortest recorded channel round trip measured in the recording interval, in microseconds.
Identifier:	MQIAMO_NET_TIME_MIN.
Data type:	MQCFIN.
Returned:	When available.

NetTimeAvg

Description:	The average recorded channel round trip measured in the recording interval, in microseconds.
Identifier:	MQIAMO_NET_TIME_AVG.
Data type:	MQCFIN.
Returned:	When available.

NetTimeMax

Description:	The longest recorded channel round trip measured in the recording interval, in microseconds.
Identifier:	MQIAMO_NET_TIME_MAX.
Data type:	MQCFIN.
Returned:	When available.

ExitTimeMin

Description:	The shortest recorded time, in microseconds, spent executing a user exit in the recording interval,
Identifier:	MQIAMO_EXIT_TIME_MIN.
Data type:	MQCFIN.
Returned:	When available.

ExitTimeAvg

Description:	The average recorded time, in microseconds, spent executing a user exit in the recording interval. Measured in microseconds.
Identifier:	MQIAMO_EXIT_TIME_AVG.
Data type:	MQCFIN.
Returned:	When available.

ExitTimeMax

Description:	The longest recorded time, in microseconds, spent executing a user exit in the recording interval. Measured in microseconds.
Identifier:	MQIAMO_EXIT_TIME_MAX.
Data type:	MQCFIN.
Returned:	When available.

FullBatchCount

Description:	The number of batches processed by the channel that were sent because the value of the channel attributes BATCHSZ or BATCHLIM was reached.
Identifier:	MQIAMO_FULL_BATCHES.
Data type:	MQCFIN.
Returned:	When available.

IncmplBatchCount

Description:	The number of batches processed by the channel, that were sent without the value of the channel attribute BATCHSZ being reached.
Identifier:	MQIAMO_INCOMPLETE_BATCHES.
Data type:	MQCFIN.
Returned:	When available.

AverageBatchSize

Description:	The average batch size of batches processed by the channel.
Identifier:	MQIAMO_AVG_BATCH_SIZE.
Data type:	MQCFIN.
Returned:	When available.

PutRetryCount

Description:	The number of times in the time interval that a message failed to be put, and entered a retry loop.
Identifier:	MQIAMO_PUT_RETRIES.
Data type:	MQCFIN.
Returned:	When available.

Reference notes

Use this page to view the notes to which descriptions of the structure of accounting and statistics messages refer

The following message data descriptions refer to these notes:

1. This parameter relates to WebSphere MQ objects. This parameter is an array of values (MQCFIL or MQCFIL64) indexed by the following constants:

Table 24. Array indexed by object type

Object type	Value context
MQOT_Q (1)	Contains the value relating to queue objects.
MQOT_NAMELIST (2)	Contains the value relating to namelist objects.

Table 24. Array indexed by object type (continued)	
Object type	**Value context**
MQOT_PROCESS (3)	Contains the value relating to process objects.
MQOT_Q_MGR (5)	Contains the value relating to queue manager objects.
MQOT_CHANNEL (6)	Contains the value relating to channel objects.
MQOT_AUTH_INFO (7)	Contains the value relating to authentication information objects.
MQOT_TOPIC (8)	Contains the value relating to topic objects.

Note: An array of 13 MQCFIL or MQCFIL64 values are returned but only those listed are meaningful.

2. This parameter relates to WebSphere MQ messages. This parameter is an array of values (MQCFIL or MQCFIL64) indexed by the following constants:

Table 25. Array indexed by persistence value	
Constant	**Value**
1	Contains the value for nonpersistent messages.
2	Contains the value for persistent messages.

Note: The index for each of these arrays starts at zero, so an index of 1 refers to the second row of the array. Elements of these arrays not listed in these tables contain no accounting or statistics information.

Application activity trace

Application activity trace produces detailed information about the behavior of applications connected to a queue manager. It traces the behavior of an application and provides a detailed view of the parameters used by an application as it interacts with IBM WebSphere MQ resources. It also shows the sequence of MQI calls issued by an application.

Use Application activity trace when you require more information than is provided by Event monitoring, Message monitoring, Accounting and statistics messages, and Real-time monitoring.

Collecting application activity trace information

An application activity trace message is a PCF message. You configure activity trace using a configuration file. To collect application activity trace information you set the ACTVTRC queue manager attribute. You can override this setting at connection level using MQCONNX options, or at application stanza level using the activity trace configuration file.

About this task

Activity trace messages are composed of an MQMD structure: a PCF (MQCFH) header structure, followed by a number of PCF parameters. A sequence of ApplicationTraceData PCF groups follows the PCF parameters. These PCF groups collect information about the MQI operations that an application performs while connected to a queue manager. You configure activity trace using a configuration file called `mqat.ini`.

To control whether or not application activity trace information is collected, you configure one or more of the following settings:

1. The ACTVTRC queue manager attribute.
2. The ACTVCONO settings (in the MQCNO structure passed in MQCONNX).
3. The matching stanza for the application in the activity trace configuration file `mqat.ini`.

The previous sequence is significant. The ACTVTRC attribute is overridden by the ACTVCONO settings, which are overridden by the settings in the `mqat.ini` file.

Trace entries are written after each operation has completed, unless otherwise stated. These entries are first written to the system queue SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE, then written to application activity trace messages when the application disconnects from the queue manager. For long running applications, intermediate messages are written if any of the following events occurs:

- The lifetime of the connection reaches a defined timeout value.
- The number of operations reaches a specified number.
- The amount of data collected in memory reaches the maximum message length allowed for the queue.

You set the timeout value using the `ActivityInterval` parameter. You set the number of operations using the `ActivityCount` parameter. Both parameters are specified in the activity trace configuration file `mqat.ini`.

Enabling application activity trace can affect performance. The overhead can be reduced by tuning the **ActivityCount** and the **ActivityInterval** settings. See "Tuning the performance impact of application activity trace" on page 203.

The simplest way to view the contents of application activity trace messages is to use the "amqsact sample program" on page 203.

Procedure

1. "Setting ACTVTRC to control collection of activity trace information" on page 196.
2. "Setting MQCONNX options to control collection of activity trace information" on page 197.
3. "Configuring activity trace behavior using mqat.ini" on page 197.
4. "Tuning the performance impact of application activity trace" on page 203.

Setting ACTVTRC to control collection of activity trace information

Use the queue manager attribute ACTVTRC to control the collection of MQI application activity trace information

About this task

Application activity trace messages are generated only for connections that begin after application activity trace is enabled. The **ACTVTRC** parameter can have the following values:

ON
    API activity trace collection is switched on

OFF
    API activity trace collection is switched off

Note: The **ACTVTRC** setting can be overridden by the queue manager **ACTVCONO** parameter. If you set the **ACTVCONO** parameter to ENABLED, then the **ACTVTRC** setting can be overridden for a given connection using the **Options** field in the MQCNO structure. See "Setting MQCONNX options to control collection of activity trace information" on page 197.

Example

To change the value of the **ACTVTRC** parameter, you use the MQSC command ALTER QMGR. For example, to enable MQI application activity trace information collection use the following MQSC command:

```
ALTER QMGR ACTVTRC(ON)
```

What to do next

The simplest way to view the contents of application activity trace messages is to use the "amqsact sample program" on page 203.

Enabling application activity trace can affect performance. The overhead can be reduced by tuning the **ActivityCount** and the **ActivityInterval** settings. See "Tuning the performance impact of application activity trace" on page 203.

Setting MQCONNX options to control collection of activity trace information

If the queue manager attribute **ACTVCONO** is set to ENABLED, you can use the **ConnectOpts** parameter on the MQCONNX call to enable or disable application activity reports on a per connection basis. These options override the activity trace behavior defined by the queue manager attribute **ACTVTRC**, and can be overridden by settings in the activity trace configuration file mqat.ini.

Procedure

1. Set the queue manager attribute **ACTVCONO** to ENABLED.

   Note: If an application attempts to modify the accounting behavior of an application using the **ConnectOpts** parameter, and the QMGR attribute **ACTVCONO** is set to DISABLED, then no error is returned to the application, and activity trace collection is defined by the queue manager attributes or the activity trace configuration file mqat.ini.

2. Set the **ConnectOpts** parameter on the MQCONNX call to MQCNO_ ACTIVITY_ TRACE_ENABLED.

   The **ConnectOpts** parameter on the MQCONNX call can have the following values:

   MQCNO_ACTIVITY_ TRACE_DISABLED
   Activity trace is switched off for the connection.

   MQCNO_ ACTIVITY_ TRACE_ENABLED
   Activity trace is switched on for the connection.

   Note: If an application selects both MQCNO_ ACTIVITY_ TRACE_ENABLED and MQCNO_ACTIVITY_ TRACE_DISABLED for MQCONNX, the call fails with a reason code of MQRC_OPTIONS_ERROR.

3. Check that these activity trace settings are not being overridden by settings in the activity trace configuration file mqat.ini.

   See "Configuring activity trace behavior using mqat.ini" on page 197.

What to do next

The simplest way to view the contents of application activity trace messages is to use the "amqsact sample program" on page 203.

Enabling application activity trace can affect performance. The overhead can be reduced by tuning the **ActivityCount** and the **ActivityInterval** settings. See "Tuning the performance impact of application activity trace" on page 203.

Configuring activity trace behavior using `mqat.ini`

The activity trace behavior is configured using a configuration file called mqat.ini. This file follows the same stanza key and parameter-value pair format as the mqs.ini and qm.ini files.

About this task

▶ Linux ▶ UNIX On UNIX and Linux systems, mqat.ini is located in the queue manager data directory, which is the same location as the qm.ini file.

Windows On Windows systems, `mqat.ini` is located in the queue manager data directory `C:\Program Files\IBM\WebSphere MQ\qmgrs\queue_manager_name`. Users running applications to be traced need permission to read this file.

Note: Queue managers migrated from IBM WebSphere MQ Version 7.1 or earlier will have the `mqat.ini` file missing. In such cases, the `mqat.ini` file needs to be created manually and 660 permissions need to be set on the file.

The syntax rules for the format of the file are:

- Text beginning with a hash or semi-colon is considered to be a comment which extends to the end of the line.
- The first significant (non-comment) line must be a stanza key.
- A stanza key consists of the name of the stanza followed by a colon.
- A parameter-value pair consists of the name of a parameter followed by an equals sign and then the value.
- Only a single parameter-value pair can appear on a line. (A parameter-value must not wrap onto another line).
- Leading and trailing whitespace is ignored. There is no limit on the amount of white space between stanza names, parameter names and values, or parameter/value pairs. Line breaks are significant and not ignored
- The maximum length for any line is 2048 characters
- The stanza keys, parameter names, and constant parameter values are not case-sensitive, but the variable parameter values (**ApplName** and **DebugPath**) are case-sensitive.

Stanza keys

Two stanza key types are allowed in the configuration file: the AllActivityTrace stanza, and the ApplicationTrace stanza

AllActivityTrace stanza

The AllActivityTrace stanza defines settings for the activity trace that is applied to all IBM WebSphere MQ connections unless overridden.

Individual values in the AllActivityTrace stanza can be overridden by more specific information in an ApplicationTrace stanza.

If more than one AllActivityTrace stanza is specified then the values in the last stanza is used. Parameters missing from the chosen AllActivityTrace take default values. Parameters and values from previous AllActivityTrace stanzas are ignored

ApplicationTrace stanza

The ApplicationTrace stanza defines settings which can be applied to a specific name, type or both of IBM WebSphere MQ connection.

This stanza includes ApplName and ApplClass values which are used according to the matching rules defined in Connection Matching Rules to determine whether the stanza applies to a particular connection.

Parameter/Value Pairs

The following table lists the parameter/value pairs which may be used in the activity trace configuration file.

Table 26. Parameter/value pairs that can be used in the activity trace configuration file			
Name	**Stanza Type**	**Values (default in bold type)**	**Description**
Trace	ApplicationTrace	ON / **OFF**	Activity trace switch. This switch can be used in the application-specific stanza to determine whether activity trace is active for the scope of the current application stanza. Note that this value overrides ACTVTRC and ACTVCONO settings for the queue manager.
ActivityInterval	AllActivityTrace ApplicationTrace	**0**-99999999 (**0=off**)	Time interval in seconds between trace messages. Activity trace does not use a timer thread, so the trace message will not be written at the exact instant that the time elapses - rather it will be written when the first MQI operation is executed after the time interval has elapsed. If this value is 0 then the trace message is written when the connection disconnects (or when the activity count is reached).
ActivityCount	AllActivityTrace ApplicationTrace	**0**-99999999 (**0=off**)	Number of MQI or XA operations between trace messages. If this value is 0 then the trace message is written when the connection disconnects (or when the activity interval has elapsed).
TraceLevel	AllActivityTrace ApplicationTrace	LOW / **MEDIUM** / HIGH	Amount of parameter detail traced for each operation. The description of individual operations details which parameters are included for each trace level.

Name	Stanza Type	Values (default in bold type)	Description
TraceMessageData	AllActivityTrace ApplicationTrace	**0**- 104 857 600 (100Mb)	Amount of message data traced in bytes for MQGET, MQPUT, MQPUT1, and Callback operations
ApplName	ApplicationTrace	Character string (Required parameter - no default)	This value is used to determine which applications the ApplicationTrace stanza applies to. It is matched to the ApplName value from the API exit context structure (which is equivalent to the MQMD.PutApplName). The content of the ApplName value varies according to the application environment. For distributed platforms, only the filename portion of the MQAXC.ApplName is matched to the value in the stanza. Characters to the left of the rightmost path separator are ignored when the comparison is made. For z/OS applications, the entire MQAXC.ApplName is matched to the value in the stanza. A single wildcard character (*) can be used at the end of the ApplName value to match any number of characters after that point are. If the ApplName value is set to a single wildcard character (*) then the ApplName value matches all applications.

Table 26. Parameter/value pairs that can be used in the activity trace configuration file (continued)

Table 26. Parameter/value pairs that can be used in the activity trace configuration file (continued)			
Name	**Stanza Type**	**Values (default in bold type)**	**Description**
ApplClass	ApplicationTrace	**USER** / MCA / INTERNAL / ALL	The class of application. See the following table for an explanation of how the AppType values correspond to IBM WebSphere MQ connections

The following table shows how the AppClass values correspond to the APICallerType and APIEnvironment fields in the connection API exit context structure.

Table 27. Appclass values and how they correspond to the APICallerType and APIEnvironment fields			
APPLCLASS	**API Caller Type:**	**API Environment:**	**Description**
USER	MQXACT_EXTERNAL	MQXE_OTHER	Only user applications are traced
MCA	(Any value)	MQXE_MCA MQXE_MCA_CLNTCONN MQXE_MCA_SVRCONN	Clients and channels (amqrmppa)
INTERNAL	MQXACT_EXTERNAL	MQXE_COMMAND_SERVER MQXE_MQSC	'runmqsc' and command server
INTERNAL	MQXACT_INTERNAL	(Any value)	"trusted" and internal applications and processes; for example, amqzdmaa
ALL	(Any value)	(Any value)	All user and internal connections are traced

> ⚠️ **Attention:** You must use an **APPLCLASS** of *MCA* for client user applications, as a class of *USER* does not match these.

For example, to trace the **amqsputc** sample application, you could use the following code:

```
ApplicationTrace:
ApplClass=MCA                           # Application type
                                        #    Values:  (USER | MCA | INTERNAL | ALL)
                                        #    Default: USER
ApplName=amqsputc      # Application name (may be wildcarded)
                                        #    (matched to app name without path)
                                        #    Default: *
Trace=ON                                # Activity trace switch for application
                                        #    Values:  ( ON | OFF )
                                        #    Default: OFF
ActivityInterval=30                      # Time interval between trace messages
                                        #    Values: 0-99999999 (0=off)
                                        #    Default: 0
ActivityCount=1                         # Number of operations between trace msgs
                                        #    Values: 0-99999999  (0=off)
                                        #    Default: 0
TraceLevel=MEDIUM                       # Amount of data traced for each operation
                                        #    Values: LOW | MEDIUM | HIGH
                                        #    Default: MEDIUM
TraceMessageData=1000                   # Amount of message data traced
                                        #    Values: 0-100000000
                                        #    Default: 0
```

Connection Matching Rules

The queue manager applies the following rules to determine which stanzas settings to use for a connection.

1. A value specified in the AllActivityTrace stanza is used for the connection unless the value also occurs in an ApplicationTrace stanza and the stanza fulfills the matching criteria for the connection described in points 2, 3, and 4.

2. The ApplClass is matched against the type of the IBM WebSphere MQ connection. If the ApplClass does not match the connection type then the stanza is ignored for this connection.

3. The ApplName value in the stanza is matched against the file name portion of the ApplName field from the API exit context structure (MQAXC) for the connection. The file name portion is derived from the characters to the right of the final path separator (/ or \) character. If the stanza ApplName includes a wildcard (*) then only the characters to the left of the wildcard are compared with the equivalent number of characters from the connections ApplName. For example, if a stanza value of "FRE*" is specified then only the first three characters are used in the comparison, so "path/FREEDOM" and "path\FREDDY" match, but "path/FRIEND" does not. If the stanzas ApplName value does not match the connection ApplName then the stanza is ignored for this connection.

4. If more than one stanza matches the connections ApplName and ApplClass, then the stanza with the most specific ApplName is used. The most specific ApplName is defined as the one which uses the most characters to match the connections ApplName. For example, if the ini file contains a stanza with ApplName="FRE*" and another stanza with ApplName="FREE*" then the stanza with ApplName="FREE*" is chosen as the best match for a connection with ApplName="path/FREEDOM" because it matches four characters (whereas ApplName="FRE*" matches only three).

5. If after applying the rules in points 2, 3, and 4, there is more than one stanza that matches the connections ApplName and ApplClass, then the values from the last matching will be used and all other stanzas will be ignored.

Application Activity Trace File Example

The following example shows how the configuration data is specified in the Activity Trace ini file. This example is shipped as a sample called `mqat.ini` in the C samples directory (the same directory as the `amqsact.c file`)

```
AllActivityTrace:
  ActivityInterval=0                  # Time interval between trace messages
                                      #   Values: 0-99999999 (0=off)
                                      #   Default: 0
  ActivityCount=0                     # Number of operations between trace msgs
                                      #   Values: 0-99999999  (0=off)
                                      #   Default: 0
  TraceLevel=MEDIUM                   # Amount of data traced for each operation
                                      #   Values: LOW | MEDIUM | HIGH
                                      #   Default: MEDIUM
  TraceMessageData=0                  # Amount of message data traced
                                      #   Values: 0-100000000
                                      #   Default: 0

ApplicationTrace:
  ApplClass=USER                       # Application type
                                      #   Values:  (USER | MCA | INTERNAL | ALL)
                                      #   Default: USER
  ApplName=AppName*                   # Application name (may be wildcarded)
                                      #   (matched to app name without path)
                                      #   Default: *
  Trace=OFF                           # Activity trace switch for application
                                      #   Values:  ( ON | OFF )
                                      #   Default: OFF
ActivityInterval=0                    # Time interval between trace messages
                                      #   Values: 0-99999999 (0=off)
                                      #   Default: 0
  ActivityCount=0                     # Number of operations between trace msgs
                                      #   Values: 0-99999999  (0=off)
                                      #   Default: 0
  TraceLevel=MEDIUM                   # Amount of data traced for each operation
                                      #   Values: LOW | MEDIUM | HIGH
                                      #   Default: MEDIUM
  TraceMessageData=0                  # Amount of message data traced
                                      #   Values: 0-100000000
                                      #   Default: 0
```

What to do next

Enabling application activity trace can affect performance. The overhead can be reduced by tuning the **ActivityCount** and the **ActivityInterval** settings. See "Tuning the performance impact of application activity trace" on page 203.

Tuning the performance impact of application activity trace

Enabling application activity trace can incur a performance penalty. This can be reduced by only tracing the applications that you need, by increasing the number of applications draining the queue, and by tuning **ActivityInterval**, **ActivityCount** and **TraceLevel** in mqat.ini.

About this task

Enabling application activity trace selectively for an application or for all queue manager applications can result in additional messaging activity, and in the queue manager requiring additional storage space. In environments where messaging performance is critical, for example, in high workload applications or where a service level agreement (SLA) requires a minimum response time from the messaging provider, it might not be appropriate to collect application activity trace or it might be necessary to adjust the detail or frequency of trace activity messages that are produced. The preset values of **ActivityInterval**, **ActivityCount** and **TraceLevel** in the mqat.ini file give a default balance of detail and performance. However, you can tune these values to meet the precise functional and performance requirements of your system.

Procedure

- Only trace the applications that you need.

  Do this by creating an ApplicationTrace application-specific stanza in mqat.ini, or by changing the application to specify MQCNO_ACTIVITY_TRACE_ENABLED in the options field on the **MQCNO** structure on an MQCONNX call. See "Configuring activity trace behavior using mqat.ini" on page 197 and "Setting MQCONNX options to control collection of activity trace information" on page 197.

- Before starting trace, check that at least one application is running and is ready to retrieve the activity trace message data from the SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE.

- Keep the queue depth as low as possible, by increasing the number of applications draining the queue.

- Set the **TraceLevel** value in the mqat.ini file to collect the minimum amount of data required.

  TraceLevel=LOW has the lowest impact to messaging performance. See "Configuring activity trace behavior using mqat.ini" on page 197.

- Tune the **ActivityCount** and **ActivityInterval** values in mqat.ini, to adjust how often activity trace messages are generated.

  If you are tracing multiple applications, the activity trace messages might be being produced faster than they can be removed from the SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE. However, when you reduce how often activity trace messages are generated, you are also increasing the storage space required by the queue manager and the size of the messages when they are written to the queue.

What to do next

amqsact sample program

amqsact formats Application Activity Trace messages for you and is provided with WebSphere MQ.

The compiled program is located in the samples directory:

- On UNIX and Linux MQ_INSTALLATION_PATH/samp/bin
- On Windows MQ_INSTALLATION_PATH\tools\c\Samples\Bin

Display mode

By default, **amqsact** in display mode processes messages on SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE. You can override this behavior by specifying a queue name or topic string.

You can also control the trace period displayed and specify whether the activity trace messages are removed or retained after display.
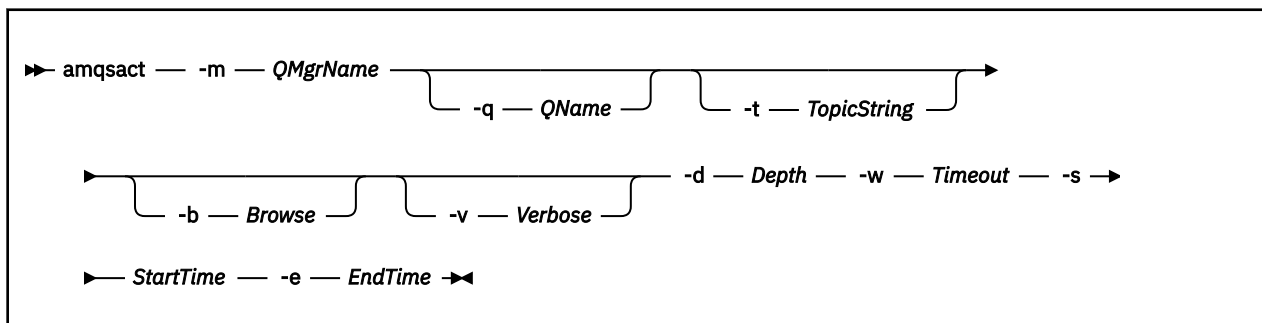
Required parameters for display mode

-m *QMgrName*
Name of the queue manager.

-d *Depth*
Number of records to display.

-w *Timeout*
Time to wait, in seconds. If no trace messages appear in the specified period, **amqsact** exits.

-s *StartTime*
Start time of record to process.

-e *EndTime*
End time of record to process.

Optional parameters for display mode

-q *QName*
Specify a specific queue to override the default queue name

-t *TopicString*
Subscribe to an event topic

-b
Browse records only

-v
Verbose output

Example output for display mode

Use **amqsact** on queue manager *TESTQM*, with verbose output, on an MQCONN API call:

```
amqsact –m TESTQM -v
```

The preceding command gives the following example output:

```
MonitoringType: MQI Activity Trace
Correl_id:
00000000: 414D 5143 5445 5354 514D 2020 2020 2020 'AMQCTESTQM    '
00000010: B5F6 4251 2000 E601                '    '
QueueManager: 'TESTQM'
Host Name: 'ADMINIB-1VTJ6N1'
IntervalStartDate: '2014-03-15'
IntervalStartTime: '12:08:10'
IntervalEndDate: '2014-03-15'
```

```
IntervalEndTime: '12:08:10'
CommandLevel: 750
SeqNumber: 0
ApplicationName: 'MQ_1\bin\amqsput.exe'
Application Type: MQAT_WINDOWS_7
ApplicationPid: 14076
UserId: 'Emma_Bushby'
API Caller Type: MQXACT_EXTERNAL
API Environment: MQXE_OTHER
Application Function: ''
Appl Function Type: MQFUN_TYPE_UNKNOWN
Trace Detail Level: 2
Trace Data Length: 0
Pointer size: 4
Platform: MQPL_WINDOWS_7
MQI Operation: 0
Operation Id: MQXF_CONN
ApplicationTid: 1
OperationDate: '2014-03-15'
OperationTime: '12:08:10'
ConnectionId:
00000000: 414D 5143 5445 5354 514D 2020 2020 2020 'AMQCTESTQM    '
00000010: FFFFFFB5FFFFFFF6 4251 2000 FFFFFFE601 '          '
QueueManager: 'TESTQM'
Completion Code: MQCC_OK
Reason Code: 0
```

Application activity trace message reference

Use this page to obtain an overview of the format of application activity trace messages and the information returned in these messages

Application activity trace messages are standard IBM WebSphere MQ messages containing a message descriptor and message data. The message data contains information about the MQI operations performed by IBM WebSphere MQ applications, or information about the activities occurring in an IBM WebSphere MQ system.

Message descriptor

- An MQMD structure

Message data

- A PCF header (MQCFH)
- Application activity trace message data that is always returned
- Application activity trace message data that is operation-specific

Application activity trace message MQMD (message descriptor)

Use this page to understand the differences between the message descriptor of application activity trace messages and the message descriptor of event messages

The parameters and values in the message descriptor of application activity trace message are the same as in the message descriptor of event messages, with the following exception:

Format

Description:	Format name of message data.
Value:	**MQFMT_ADMIN** Admin message.

CorrelId

Description:	Correlation identifier.
Value:	Initialized with the ConnectionId of the application

MQCFH (PCF Header)

Use this page to view the PCF values contained by the MQCFH structure for an activity trace message

For an activity trace message, the MQCFH structure contains the following values:

Type

Description:	Structure type that identifies the content of the message.
Data type:	MQLONG.
Value:	MQCFT_APP_ACTIVITY

StrucLength

Description:	Length in bytes of MQCFH structure.
Data type:	MQLONG.
Value:	MQCFH_STRUC_LENGTH

Version

Description:	Structure version number.
Data type:	MQLONG.
Values:	MQCFH_VERSION_3

Command

Description:	Command identifier. This field identifies the category of the message.
Data type:	MQLONG.
Values:	MQCMD_ACTIVITY_TRACE

MsgSeqNumber

Description:	Message sequence number. This field is the sequence number of the message within a group of related messages.
Data type:	MQLONG.
Values:	1

Control

Description:	Control options.
Data type:	MQLONG.
Values:	MQCFC_LAST.

CompCode

Description:	Completion code.
Data type:	MQLONG.
Values:	MQCC_OK.

Reason

Description:	Reason code qualifying completion code.
Data type:	MQLONG.

Values:	MQRC_NONE.

ParameterCount

Description:	Count of parameter structures. This field is the number of parameter structures that follow the MQCFH structure. A group structure (MQCFGR), and its included parameter structures, are counted as one structure only.
Data type:	MQLONG.
Values:	1 or greater

Application activity trace message data

Immediately following the PCF header is a set of parameters describing the time interval for the activity trace. These parameters also indicate the sequence of messages in the event of messages being written. The order and number of fields following the header is not guaranteed, allowing additional information to be added in the future.

Message name:	Activity trace message.
System queue:	SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE.

QueueManager

Description:	The name of the queue manager
Identifier:	MQCA_Q_MGR_NAME
Data type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH

QSGName
HostName

Description:	The host name of the machine the Queue Manager is running on
Identifier:	MQCACF_HOST_NAME
Data type:	MQCFST

IntervalStartDate

Description:	The date of the start of the monitoring period
Identifier:	MQCAMO_START_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH

IntervalStartTime

Description:	The time of the start of the monitoring period
Identifier:	MQCAMO_START_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH

IntervalEndDate

Description:	The date of the end of the monitoring period

Identifier:	MQCAMO_END_DATE
Data type:	MQCFST
Maximum length:	MQ_DATE_LENGTH

IntervalEndTime

Description:	The time of the end of the monitoring period
Identifier:	MQCAMO_END_TIME
Data type:	MQCFST
Maximum length:	MQ_TIME_LENGTH

CommandLevel

Description:	The IBM WebSphere MQ command level
Identifier:	MQIA_COMMAND_LEVEL
Data type:	MQCFIN

SeqNumber

Description:	The sequence number normally zero. This value is incremented for each subsequent record for long running connections.
Identifier:	MQIACF_SEQUENCE_NUMBER
Data type:	MQCFIN

ApplicationName

Description:	The name of the application. (program name)
Identifier:	MQCACF_APPL_NAME
Data type:	MQCFST
Maximum length:	MQ_APPL_NAME_LENGTH

ApplClass

Description:	Type of application that performed the activity. Possible values: MQAT_*
Identifier:	MQIA_APPL_TYPE
Data type:	MQCFIN

ApplicationPid

Description:	The operating system Process ID of the application
Identifier:	MQIACF_PROCESS_ID
Data type:	MQCFIN

UserId

Description:	The user identifier context of the application
Identifier:	MQCACF_USER_IDENTIFIER
Data type:	MQCFST
Maximum length:	MQ_USER_ID_LENGTH

APICallerType

Description:	The type of the application. Possible values: MQXACT_EXTERNAL or MQXACT_INTERNAL
Identifier:	MQIACF_API_CALLER_TYPE
Data type:	MQCFIN

Environment

Description:	The runtime environment of the application. Possible values: MQXE_OTHER MQXE_MCA MQXE_MCA_SVRCONN MQXE_COMMAND_SERVER MQXE_MQSC
Identifier:	MQIACF_API_ENVIRONMENT
Data type:	MQCFIN

Detail

Description:	The detail level that is recorded for the connection. Possible values: 1=LOW 2=MEDIUM 3=HIGH
Identifier:	MQIACF_TRACE_DETAIL
Data type:	MQCFIN

TraceDataLength

Description:	The length of message data (in bytes) that is traced for this connection.
Identifier:	MQIACF_TRACE_DATA_LENGTH
Data type:	MQCFIN

Pointer Size

Description:	The length (in bytes) of pointers on the platform the application is running (to assist in interpretation of binary structures)
Identifier:	MQIACF_POINTER_SIZE
Data type:	MQCFIN

Platform

Description:	The platform on which the queue manager is running. Value is one of the MQPL_* values.
Identifier:	MQIA_PLATFORM
Data type:	MQCFIN

Variable parameters for application activity MQI operations

The application activity data MQCFGR structure is followed by the set of PCF parameters which corresponds to the operation being performed . The parameters for each operation are defined in the following section.

The trace level indicates the level of trace granularity that is required for the parameters to be included in the trace. The possible trace level values are:

1. Low

   The parameter is included when "low", "medium" or "high" activity tracing is configured for an application. This setting means that a parameter is always included in the AppActivityData group

for the operation. This set of parameters is sufficient to trace the MQI calls an application makes, and to see if they are successful.

2. Medium

   The parameter is only included in the `AppActivityData` group for the operation when "medium" or "high" activity tracing is configured for an application. This set of parameters adds information about the resources, for example, queue and topic names used by the application.

3. High

   The parameter is only included in the `AppActivityData` group for the operation when "high" activity tracing is configured for an application. This set of parameters includes memory dumps of the structures passed to the MQI and XA functions. For this reason, it contains more information about the parameters used in MQI and XA calls. The structure memory dumps are shallow copies of the structures. To avoid erroneous attempts to dereference pointers, the pointer values in the structures are set to NULL.

   Note: The version of the structure that is dumped is not necessarily identical to the version used by an application. The structure can be modified by an API crossing exit, by the activity trace code, or by the queue manager. A queue manager can modify a structure to a later version, but the queue manager never changes it to an earlier version of the structure. To do so, would risk losing data.

MQBACK
Application has started the MQBACK MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type	MQCFIN

MQBEGIN
Application has started the MQBEGIN MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type	MQCFIN

MQBO

Description:	The MQBEGIN options structure. This parameter is not included if a NULL pointer is used on the MQBEGIN call.
PCF Parameter:	MQBACF_MQBO_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQBO structure.

MQCALLBACK

Application has started the MQCALLBACK function

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type	MQCFIN

CallType

Description:	Why function has been called. One of the MQCBCT_* values
PCF Parameter:	MQIACF_CALL_TYPE
Trace level:	1
Type	MQCFIN

MsgBuffer

Description:	Message data.
PCF Parameter:	MQBACF_MESSAGE_DATA
Trace level:	1
Type	MQCFBS
Length:	Length is governed by the TRACEDATA() parameter set in the APPTRACE configuration. If TRACEDATA=NONE then this parameter is omitted.

MsgLength

Description:	Length of the message. (Taken from the DataLength field in the MQCBC structure).
PCF Parameter:	MQIACF_MSG_LENGTH
Trace level:	1
Type	MQCFIN

HighResTime

Description:	Time of operation in microseconds since midnight, January 1st 1970 (UTC)
	Note: The accuracy of this timer varies according to platform support for high a resolution timer
PCF Parameter:	MQIAMO64_HIGHRES_TIME

Trace level: 2

Type MQCFIN64

ReportOptions

Description: Options for report messages

PCF Parameter: MQIACF_REPORT

Trace level: 2

Type MQCFIN

MsgType

Description: Type of message

PCF Parameter: MQIACF_MSG_TYPE

Trace level: 2

Type MQCFIN

Expiry

Description: Message lifetime

PCF Parameter: MQIACF_EXPIRY

Trace level: 2

Type MQCFIN

Format

Description: Format name of message data

PCF Parameter: MQCACH_FORMAT_NAME

Trace level: 2

Type MQCFST

Length: MQ_FORMAT_LENGTH

Priority

Description: Message priority

PCF Parameter: MQIACF_PRIORITY

Trace level: 2

Type MQCFIN

Persistence

Description: Message persistence

PCF Parameter: MQIACF_PERSISTENCE

Trace level: 2

Type MQCFIN

MsgId

Description: Message identifier

PCF Parameter:	MQBACF_MSG_ID
Trace level:	2
Type	MQCFBS
Length:	MQ_MSG_ID_LENGTH

CorrelId

Description:	Correlation identifier
PCF Parameter:	MQBACF_CORREL_ID
Trace level:	2
Type	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

ObjectName

Description:	The name of the opened object.
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH

ResolvedQName

Description:	The local name of the queue from which the message was retrieved.
PCF Parameter:	MQCACF_RESOLVED_Q_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH

ReplyToQueue

Description:	MQ_Q_NAME_LENGTH
PCF Parameter:	MQCACF_REPLY_TO_Q
Trace level:	2
Type	MQCFST

ReplyToQMgr

Description:	MQ_Q_MGR_NAME_LENGTH
PCF Parameter:	MQCACF_REPLY_TO_Q_MGR
Trace level:	2
Type	MQCFST

CodedCharSetId

Description:	Character set identifier of message data
PCF Parameter:	MQIA_CODED_CHAR_SET_ID

Trace level:	2
Type	MQCFIN

Encoding

Description:	Numeric encoding of message data.
PCF Parameter:	MQIACF_ENCODING
Trace level:	2
Type	MQCFIN

PutDate

Description:	MQ_PUT_DATE_LENGTH
PCF Parameter:	MQCACF_PUT_DATE
Trace level:	2
Type	MQCFST

PutTime

Description:	MQ_PUT_TIME_LENGTH
PCF Parameter:	MQCACF_PUT_TIME
Trace level:	2
Type	MQCFST

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_LOCAL _Q_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH.

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2

Type	MQCFIN

PolicyName

Description:	The policy name that was applied to this message.
	Note: AMS protected messages only
PCF Parameter:	MQCA_POLICY_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_OBJECT_NAME_LENGTH

XmitqMsgId

Description:	The message ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_MSG_ID
Trace level:	2
Type	MQCFBS
Length:	MQ_MSG_ID_LENGTH

XmitqCorrelId

Description:	The correlation ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_CORREL_ID
Trace level:	2
Type	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

XmitqPutTime

Description:	The put time of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_TIME
Trace level:	2
Type	MQCFST
Length:	MQ_PUT_TIME_LENGTH

XmitqPutDate

Description:	The put date of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_DATE
Trace level:	2
Type	MQCFST

Length:	MQ_PUT_DATE_LENGTH

XmitqRemoteQName

Description:	The remote queue destination of the message in the transmission queue header. **Note:** Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_Name
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH

XmitqRemoteQMgr

Description:	The message ID of the message in the transmission queue header. **Note:** Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_MGR
Trace level:	2
Type	MQCFST
Length:	MQ_MSG_ID_LENGTH

MsgDescStructure

Description:	The MQMD structure. This parameter is omitted if a version 4 MQGMO was used to request that a Message Handle be returned instead of an MQMD
PCF Parameter:	MQBACF_MQMD_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQMD structure (actual size is dependent on structure version)

GetMsgOptsStructure

Description:	The MQGMO structure.
PCF Parameter:	MQBACF_MQGMO_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQGMO structure (actual size is dependent on structure version)

MQCBContextStructure

Description:	The MQCBC structure.
PCF Parameter:	MQBACF_MQCBC_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQCBC structure (actual size is dependent on structure version)

MQCB

Application has started the manage callback MQI function

CallbackOperation

Description:	The manage callback function operation. Set to one of the MQOP_* values
PCF Parameter:	MQIACF_MQCB_OPERATION
Trace level:	1
Type	MQCFIN

CallbackType

Description:	The type of the callback function (CallbackType field from the MQCBD structure). Set to one of the MQCBT_* values
PCF Parameter:	MQIACF_MQCB_TYPE
Trace level:	1
Type	MQCFIN

CallbackOptions

Description:	The callback options. Set to one of the MQCBDO_* values
PCF Parameter:	MQIACF_MQCB_OPTIONS
Trace level:	1
Type	MQCFIN

CallbackFunction

Description:	The pointer to the callback function if started as a function call.
PCF Parameter:	MQBACF_MQCB_FUNCTION
Trace level:	1
Type	MQCFBS
Length:	Size of MQPTR

CallbackName

Description:	The name of the callback function if started as a dynamically linked program.
PCF Parameter:	MQCACF_MQCB_NAME
Trace level:	1
Type	MQCFST
Length:	Size of MQCHAR128

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type	MQCFIN

MaxMsgLength

Description:	Maximum message length. Set to an integer, or the special value MQCBD_FULL_MSG_LENGTH
PCF Parameter:	MQIACH_MAX_MSG_LENGTH
Trace level:	2
Type	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type	MQCFIN

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH.

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type	MQCFIN

CallBack DescriptorStructure

Description:	The MQCBD structure. This parameter is omitted if a NULL MQCBC value is passed to the MQCB call.
PCF Parameter:	MQBACF_MQCBD_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQCBC structure

MsgDescStructure

Description:	The MQMD structure. The MsgDescStructure parameter is omitted if a NULL MQMD value is passed to the MQCB call.
PCF Parameter:	MQBACF_MQMD_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQMD structure (actual size depends on structure version)

GetMsgOptsStructure

Description:	The MQGMO structure. This parameter is omitted if a NULL MQGMO value is passed to the MQCB call.
PCF Parameter:	MQBACF_MQGMO_STRUCT
Trace level:	3
Type	MQCFBS
Length:	The length in bytes of the MQGMO structure (actual size depends on structure version)

MQCLOSE
Application has started the MQCLOSE MQI function

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type	MQCFIN

CloseOptions

Description:	Close options
PCF Parameter:	MQIACF_CLOSE_OPTIONS
Trace level:	1
Type	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE

Trace level:	1
Type	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type	MQCFIN

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH.

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type	MQCFIN

MQCMIT
Application has started the MQCMIT MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type	MQCFIN

Reason

Description:	The reason code result of the operation

PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type	MQCFIN

MQCONN and MQCONNX

Application has started the MQCONN or MQCONNX MQI function

ConnectionId

Description:	The Connection ID if available or MQCONNID_NONE if not
PCF Parameter:	MQBACF_CONNECTION_ID
Trace level:	1
Type:	MQCFBS
Maximum length:	MQ_CONNECTION_ID_LENGTH

QueueManagerName

Description:	The (unresolved) name of the queue manager used in the MQCONN(X) call
PCF Parameter:	MQCA_Q_MGR_NAME
Trace level:	1
Type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

ConnectOptions

Description:	Connect Options Derived from MQCNO_* values
	Note: MQCONNX only
PCF Parameter:	MQIACF_CONNECT_OPTIONS
Trace level:	2
Type:	MQCFIN

ConnectionOptionsStructure

Description:	The MQCNO structure.
	Note: MQCONNX only)
PCF Parameter:	MQBACF_MQCNO_STRUCT
Trace level:	3
Type:	MQCFBS
Maximum length:	The length in bytes of the MQCNO structure (actual size depends on structure version)

ChannelDefinitionStructure

Description:	The MQCD structure.
	Note: Client connections only
PCF Parameter:	MQBACF_MQCD_STRUCT
Trace level:	3
Type:	MQCFBS
Maximum length:	The length in bytes of the MQCD structure (actual size depends on structure version)

MQCTL

Application has started the MQCTL MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

CtlOperation

Description:	One of MQOP_* values
PCF Parameter:	MQIACF_CTL_OPERATION
Trace level:	1
Type:	MQCFIN

MQDISC
Application has started the MQDISC MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

MQGET
Application has started the MQGET MQI function

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type:	MQCFIN

GetOptions

Description:	The get options from MQGMO.Options
PCF Parameter:	MQIACF_GET_OPTIONS
Trace level:	1
Type:	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

MsgBuffer

Description:	Message data. If TRACEDATA=NONE then this parameter is omitted

PCF Parameter:	MQBACF_MESSAGE_DATA
Trace level:	1
Type:	MQCFBS
Maximum length:	Length is governed by the TRACEDATA() parameter set in the APPTRACE configuration. (Included in the trace message as MQIACF_TRACE_DATA_LENGTH).

MsgLength

Description:	Length of the message.
PCF Parameter:	MQIACF_MSG_LENGTH
Trace level:	1
Type:	MQCFIN

HighResTime

Description:	Time of operation in microseconds since midnight, January 1 1970 (UTC)
	Note: The accuracy of this timer varies according to platform support for high a resolution timer
PCF Parameter:	MQIAMO64_HIGHRES_TIME
Trace level:	2
Type:	MQCFIN64

BufferLength

Description:	Length of the buffer provided by the application
PCF Parameter:	MQIACF_BUFFER_LENGTH
Trace level:	2
Type:	MQCFIN

ObjectName

Description:	The name of the opened object
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

ResolvedQName

Description:	The local name of the queue from which the message was retrieved.
PCF Parameter:	MQCACF_RESOLVED_Q_NAME
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_Q_NAME_LENGTH

ReportOptions

Description:	Message report options
PCF Parameter:	MQIACF_REPORT
Trace level:	2
Type:	MQCFIN

MsgType

Description:	Type of message
PCF Parameter:	MQIACF_MSG_TYPE
Trace level:	2
Type:	MQCFIN

Expiry

Description:	Message lifetime
PCF Parameter:	MQIACF_EXPIRY
Trace level:	2
Type:	MQCFIN

Format

Description:	Format name of message data
PCF Parameter:	MQCACH_FORMAT_NAME
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_FORMAT_LENGTH

Priority

Description:	Message priority
PCF Parameter:	MQIACF_PRIORITY
Trace level:	2
Type:	MQCFIN

Persistence

Description:	Message persistence
PCF Parameter:	MQIACF_PERSISTENCE
Trace level:	2
Type:	MQCFIN

MsgId

Description:	Message identifier
PCF Parameter:	MQBACF_MSG_ID
Trace level:	2

Type:	MQCFBS
Maximum length:	MQ_MSG_ID_LENGTH

CorrelId

Description:	Correlation identifier
PCF Parameter:	MQBACF_CORREL_ID
Trace level:	2
Type:	MQCFBS
Maximum length:	MQ_CORREL_ID_LENGTH

ReplyToQueue

Description:	
PCF Parameter:	MQCACF_REPLY_TO_Q
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_Q_NAME_LENGTH

ReplyToQMgr

Description:	
PCF Parameter:	MQCACF_REPLY_TO_Q_MGR
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH

CodedCharSetId

Description:	Character set identifier of message data
PCF Parameter:	MQIA_CODED_CHAR_SET_ID
Trace level:	2
Type:	MQCFIN

Encoding

Description:	Numeric encoding of message data.
PCF Parameter:	MQIACF_ENCODING
Trace level:	2
Type:	MQCFIN

PutDate

Description:	
PCF Parameter:	MQCACF_PUT_DATE
Trace level:	2
Type:	MQCFST

Maximum length: MQ_PUT_DATE_LENGTH

PutTime

Description:

PCF Parameter: MQCACF_PUT_TIME

Trace level: 2

Type: MQCFST

Maximum length: MQ_PUT_TIME_LENGTH

ResolvedQName

Description: The queue name referred to by the ObjectHandle, when ResolvedType is
                   MQOT_Q.

PCF Parameter: MQCACF_RESOLVED_LOCAL_Q_NAME

Trace level: 2

Type   MQCFST

Length: MQ_Q_NAME_LENGTH.

ResObjectString

Description: The object name referred to by the ObjectHandle, when ResolvedType is
                   MQOT_TOPIC.

PCF Parameter: MQCACF_RESOLVED_OBJECT_STRING

Trace level: 2

Type   MQCFST

Length: Length varies.

ResolvedType

Description: The type of the object referred to by the ObjectHandle. Possible values are
                   MQOT_Q, MQOT_TOPIC, or MQOT_NONE.

PCF Parameter: MQIACF_RESOLVED_TYPE

Trace level: 2

Type   MQCFIN

PolicyName

Description: The policy name that was applied to this message.

                   Note: AMS protected messages only

PCF Parameter: MQCA_POLICY_NAME

Trace level: 2

Type: MQCFST

Length: MQ_OBJECT_NAME_LENGTH

XmitqMsgId

Description:	The message ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

XmitqCorrelId

Description:	The correlation ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

XmitqPutTime

Description:	The put time of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_TIME
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_TIME_LENGTH

XmitqPutDate

Description:	The put date of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_DATE
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_DATE_LENGTH

XmitqRemoteQName

Description:	The remote queue destination of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

XmitqRemoteQMgr

Description:	The remote queue manager destination of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

MsgDescStructure

Description:	The MQMD structure.
PCF Parameter:	MQBACF_MQMD_STRUCT
Trace level:	3
Type:	MQCFBS
Maximum length:	The length in bytes of the MQMD structure (actual size depends on structure version)

GetMsgOptsStructure

Description:	The MQGMO structure.
PCF Parameter:	MQBACF_MQGMO_STRUCT
Trace level:	3
Type:	MQCFBS
Maximum length:	The length in bytes of the MQGMO structure (actual size depends on structure version)

MQINQ

Application has started the MQINQ MQI function

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type:	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE

Trace level:	1	
Type:	MQCFIN	

SelectorCount

Description:	The count of selectors that are supplied in the Selectors array.
PCF Parameter:	MQIACF_SELECTOR_COUNT
Trace level:	2
Type:	MQCFIN

Selectors

Description:	The list of attributes (integer or character) whose values must be returned by MQINQ.
PCF Parameter:	MQIACF_SELECTORS
Trace level:	2
Type:	MQCFIL

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_Q_NAME
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_Q_NAME_LENGTH

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type:	MQCFST
Maximum length:	Length varies

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type:	MQCFIN

IntAttrCount

Description:	The number of integer attributes returned by the inquire operation
PCF Parameter:	MQIACF_INTATTR_COUNT
Trace level:	3

Type:	MQCFIN

IntAttrs

Description:	The integer attribute values returned by the inquire operation. This parameter is only present if IntAttrCount is > 0 when MQINQ returns.
PCF Parameter:	MQIACF_INT_ATTRS
Trace level:	3
Type:	MQCFIL

CharAttrs

Description:	The character attributes returned by the inquire operation. The values are concatenated together. This parameter is only included if CharAttrLength is > 0 when MQINQ returns.
PCF Parameter:	MQCACF_CHAR_ATTRS
Trace level:	3
Type:	MQCFST

MQOPEN

Application has started the MQOPEN MQI function

ObjectType

Description:	The object type passed in MQOT.ObjectType
PCF Parameter:	MQIACF_OBJECT_TYPE
Trace level:	1
Type:	MQCFIN

ObjectName

Description:	The name of the object passed to the MQI call before any queue name resolution is attempted.
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	1
Type:	MQCFST
Maximum length:	MQ_Q_NAME_LENGTH

ObjectQMgrName

Description:	The name of the object queue manager passed to the MQI call before any queue name resolution is attempted.
PCF Parameter:	MQCACF_OBJECT_Q_MGR_NAME
Trace level:	1
Type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH

ObjectHandle

Description:	The object handle

PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type:	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

OpenOptions

Description:	Options used to open the object
PCF Parameter:	MQIACF_OPEN_OPTIONS
Trace level:	1
Type:	MQCFIN

AlternateUserId

Description:	Only included if MQOO_ALTERNATE_USER_AUTHORITY is specified
PCF Parameter:	MQCACF_ALTERNATE_USERID
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_USER_ID_LENGTH

RecsPresent

Description:	The number of object name records present. Only included if MQOD Version >= MQOD_VERSION_2
PCF Parameter:	MQIACF_RECS_PRESENT
Trace level:	1
Type:	MQCFIN

KnownDestCount

Description:	Number of local queues opened successfully Only included if MQOD Version >= MQOD_VERSION_2
PCF Parameter:	MQIACF_KNOWN_DEST_COUNT
Trace level:	1
Type:	MQCFIN

UnknownDestCount

Description:	Number of remote queues opened successfully Only included if MQOD Version >= MQOD_VERSION_2
PCF Parameter:	MQIACF_UNKNOWN_DEST_COUNT
Trace level:	1
Type:	MQCFIN

InvalidDestCount

Description:	Number of queues that failed to open Only included if MQOD Version >= MQOD_VERSION_2
PCF Parameter:	MQIACF_INVALID_DEST_COUNT
Trace level:	1
Type:	MQCFIN

DynamicQName

Description:	The dynamic queue name passed as input to the MQOPEN call.
PCF Parameter:	MQCACF_DYNAMIC_Q_NAME
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_Q_NAME_LENGTH

ResolvedLocalQName[12]

Description:	Contains the local queue name after name resolution has been carried out. (e.g. for remote queues this will be the name of the transmit queue)
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type:	MQCFST
Range:	If MQOD.Version is less than MQOD_VERSION_3 this contains the value of the MQOD.ObjectName field after the MQOPEN call has completed. If MQOD.Version is equal or greater than MQOD_VERSION_3 this contains the value in the MQOD.ResolvedQName field.
Maximum length:	MQ_Q_NAME_LENGTH

ResolvedLocalQMgrName[12]

Description:	The local queue manager name after name resolution has been performed.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_MGR
Trace level:	2
Type:	MQCFST
Range:	Only if MQOD.Version >= MQOD_VERSION_3
Maximum length:	MQ_Q_MGR_NAME_LENGTH

ResolvedQName[12]

Description:	The queue name after name resolution has been carried out.

PCF Parameter:	MQCACF_RESOLVED_Q_NAME
Trace level:	2
Type:	MQCFST
Range:	If MQOD.Version is less than MQOD_VERSION_3 this contains the value of the MQOD.ObjectName field after the MQOPEN call has completed. If MQOD.Version is equal or greater than MQOD_VERSION_3 this contains the value in the MQOD.ResolvedQName field.
Maximum length:	MQ_Q_NAME_LENGTH

ResolvedQMgrName[12]

Description:	Contains the queue manager name after name resolution has been carried out. If MQOD.Version is less than MQOD_VERSION_3 this contains the value of the MQOD. ObjectQMgrName field after the MQOPEN call has completed. If MQOD.Version is equal or greater than MQOD_VERSION_3 this contains the value in the MQOD. ResolvedQMgrName field.
PCF Parameter:	MQCACF_RESOLVED_Q_MGR
Trace level:	2
Type:	MQCFST
Maximum length:	MQ_Q_MGR_NAME_LENGTH

AlternateSecurityId

Description:	Alternative security identifier. Only present if MQOD.Version is equal or greater than MQOD_VERSION_3, MQOO_ALTERNATE_USER_AUTHORITY is specified, and MQOD.AlternateSecurityId is not equal to MQSID_NONE.
PCF Parameter:	MQBACF_ALTERNATE_SECURITYID
Trace level:	2
Type:	MQCFBS
Maximum length:	MQ_SECURITY_ID_LENGTH

ObjectString

Description:	Long object name. Only included if MQOD.Version is equal or greater than MQOD_VERSION_4 and the VSLength field of MQOD.ObjectString is MQVS_NULL_TERMINATED or greater than zero.
PCF Parameter:	MQCACF_OBJECT_STRING
Trace level:	2
Type:	MQCFST
Maximum length:	Length varies.

SelectionString

Description:	Selection string. Only included if MQOD.Version is equal or greater than MQOD_VERSION_4 and the VSLength field of MQOD. SelectionString is MQVS_NULL_TERMINATED or greater than zero.
PCF Parameter:	MQCACF_SELECTION_STRING
Trace level:	2
Type:	MQCFST

Maximum length: Length varies.

ResObjectString

Description: The long object name after the queue manager resolves the name provided in
                    the ObjectName field. Only included for topics and queue aliases that reference
                    a topic object if MQOD.Version is equal or greater than MQOD_VERSION_4 and
                    VSLength is MQVS_NULL_TERMINATED or greater than zero.

PCF Parameter: MQCACF_RESOLVED_OBJECT_STRING

Trace level: 2

Type: MQCFST

Maximum length: Length varies.

ResolvedType

Description: The type of the resolved (base) object being opened. Only included if
                    MQOD.Version is equal or greater than MQOD_VERSION_4. Possible values are
                    MQOT_Q, MQOT_TOPIC, or MQOT_NONE.

PCF Parameter: MQIACF_RESOLVED_TYPE

Trace level: 2

Type: MQCFIN

Application Activity Distribution List PCF Group Header Structure
If the MQOPEN function opens a distribution list, then the MQOPEN parameters includes one
AppActivityDistList PCF group for each of the queues in the distribution list up to the number of structures
numbered in RecsPresent. The Ap-pActivityDistList PCF group combines information from the MQOR, and
MQRR structures to identify the queue name, and indicate the result of the open operation on the queue.
An AppActivityDistList group always starts with the following MQCFGR structure:

Table 28. AppActivityDistList group MQCFGR structure

MQCFGR field	Value	Description
Type	MQCFT_GROUP	
StrucLength	Length in bytes of the MQCFGR structure	
Parameter	MQGACF_APP_DIST_LIST	Distribution list group parameter
ParameterCount	4	The number of parameter structures following the MQCFGR structure that are contained within this group.

ObjectName

Description: The name of a queue in the distribution list MQ_Q_NAME_LENGTH. Only
                    included if MQOR structures are provided.

PCF Parameter: MQCACF_OBJECT_NAME

---

[1] This parameter is only included if the object being opened resolves to a queue, and the queue is opened for
     MQOO_INPUT_*, MQOO_OUTPUT, or MQOO_BROWSE
[2] The ResolvedLocalQName parameter is only included if it is different from the ResolvedQName parameter.

Trace level:	2	
Type:	MQCFST	
Length:	MQ_Q_NAME_LENGTH. Only included if MQOR structures are provided.	

ObjectQMgrName

Description:	The name of the queue manager on which the queue named in ObjectName is defined.
PCF Parameter:	MQCACF_OBJECT_Q_MGR_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH. Only included if MQOR structures are provided.

CompCode

Description:	The completion code indicating the result of the open for this object. Only included if MQRR structures are provided and the reason code for the MQOPEN is MQRC_MULTIPLE_REASONS
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	2
Type:	MQCFIN

Reason

Description:	The reason code indicating the result of the open for this object. Only included if MQRR structures are provided and the reason code for the MQOPEN is MQRC_MULTIPLE_REASONS
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	2
Type:	MQCFIN

MQPUT
Application has started the MQPUT MQI function.

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type:	MQCFIN

PutOptions

Description:	The put options from MQPMO.Options
PCF Parameter:	MQIACF_PUT_OPTIONS
Trace level:	1
Type:	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

MsgBuffer

Description:	Message data.
PCF Parameter:	MQBACF_MESSAGE_DATA
Trace level:	1
Type:	MQCFBS
Length:	Length is governed by the TRACEDATA() parameter set in the APPTRACE configuration. If TRACEDATA=NONE then this parameter is omitted.

MsgLength

Description:	Length of the message.
PCF Parameter:	MQIACF_MSG_LENGTH
Trace level:	1
Type:	MQCFIN

RecsPresent

Description:	The number of put message records or response records present. Only included if MQPMO Version >= MQPMO_VERSION_2
PCF Parameter:	MQIACF_RECS_PRESENT
Trace level:	1
Type:	MQCFIN

KnownDestCount

Description:	Number of messages sent successfully to local queues
PCF Parameter:	MQIACF_KNOWN_DEST_COUNT
Trace level:	1
Type:	MQCFIN

UnknownDestCount

Description:	Number of messages sent successfully to remote queues
PCF Parameter:	MQIACF_UNKNOWN_DEST_COUNT

Trace level:	1
Type:	MQCFIN

InvalidDestCount

Description:	Number of messages that could not be sent
PCF Parameter:	MQIACF_INVALID_DEST_COUNT
Trace level:	1
Type:	MQCFIN

HighResTime

Description:	Time of operation in microseconds since midnight, January 1st 1970 (UTC)
	Note: The accuracy of this timer varies according to platform support for high a resolution timer.
PCF Parameter:	MQIAMO64_HIGHRES_TIME
Trace level:	2
Type:	MQCFIN64

ObjectName

Description:	The name of the opened object.
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

ResolvedQName

Description:	The name of the queue after queue name resolution has been performed.
PCF Parameter:	MQCACF_RESOLVED_Q_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

ResolvedQMgrName

Description:	The queue manager name after name resolution has been performed.
PCF Parameter:	MQCACF_RESOLVED_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH

ResolvedLocalQName[3]

Description:	Contains the local queue name after name resolution has been carried out.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME

Trace level: 2

Type: MQCFST

ResolvedLocalQMgrName[3]

Description:	Contains the local queue manager name after name resolution has been carried out.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH

ReportOptions

Description:	Message report options
PCF Parameter:	MQIACF_REPORT
Trace level:	2
Type:	MQCFIN

MsgType

Description:	Type of message
PCF Parameter:	MQIACF_MSG_TYPE
Trace level:	2
Type:	MQCFIN

Expiry

Description:	Message lifetime
PCF Parameter:	MQIACF_EXPIRY
Trace level:	2
Type:	MQCFIN

Format

Description:	Format name of message data
PCF Parameter:	MQCACH_FORMAT_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_FORMAT_LENGTH

Priority

Description:	Message priority
PCF Parameter:	MQIACF_PRIORITY
Trace level:	2
Type:	MQCFIN

Persistence

Description:	Message persistence
PCF Parameter:	MQIACF_PERSISTENCE
Trace level:	2
Type:	MQCFIN

MsgId

Description:	Message identifier
PCF Parameter:	MQBACF_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

CorrelId

Description:	Correlation identifier
PCF Parameter:	MQBACF_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

ReplyToQueue

Description:	
PCF Parameter:	MQCACF_REPLY_TO_Q
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

ReplyToQMgr

Description:	
PCF Parameter:	MQCACF_REPLY_TO_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH

CodedCharSetId

Description:	Character set identifier of message data
PCF Parameter:	MQIA_CODED_CHAR_SET_ID
Trace level:	2
Type:	MQCFIN

Encoding

Description:	Numeric encoding of message data.
PCF Parameter:	MQIACF_ENCODING
Trace level:	2
Type:	MQCFIN

PutDate

Description:	
PCF Parameter:	MQCACF_PUT_DATE
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_DATE_LENGTH

PutTime

Description:	
PCF Parameter:	MQCACF_PUT_TIME
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_TIME_LENGTH

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH.

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type:	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type:	MQCFIN

PolicyName

Description:	The policy name that was applied to this message.
	Note: AMS protected messages only
PCF Parameter:	MQCA_POLICY_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_OBJECT_NAME_LENGTH

XmitqMsgId

Description:	The message ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

XmitqCorrelId

Description:	The correlation ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

XmitqPutTime

Description:	The put time of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_TIME
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_TIME_LENGTH

XmitqPutDate

Description:	The put date of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_DATE
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_DATE_LENGTH

XmitqRemoteQName

Description:	The remote queue destination of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

XmitqRemoteQMgr

Description:	The remote queue manager destination of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

PutMsgOptsStructure

Description:	The MQPMO structure.
PCF Parameter:	MQBACF_MQPMO_STRUCT
Trace level:	3
Type:	MQCFBS
Length:	The length in bytes of the MQPMO structure (actual size depends on structure version)

MQPUT Application Activity Distribution List PCF Group Header Structure

If the MQPUT function is putting to a distribution list, then the MQPUT parameters include one AppActivityDistList PCF group. For each of the queues in the distribution list, see "Application Activity Distribution List PCF Group Header Structure" on page 235. The AppActivityDistList PCF group combines information from the MQPMR, and MQRR structures to identify the PUT parameters, and indicate the result of the PUT operation on each queue. For MQPUT operations the AppActivityDistList group contains some or all of the following parameters (the CompCode and Reason is present if the reason code is MQRC_MULTIPLE_REASONS and the other parameters are determined by the MQPMO.PutMsgRecFields field):

CompCode

Description:	The completion code indicating the result of the operation. Only included if MQRR structures are provided and the reason code for the MQPUT is MQRC_MULTIPLE_REASONS
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	2
Type:	MQCFIN

---

[3] The ResolvedLocalQName parameter is only included if it is different from the ResolvedQName parameter.

Reason

Description:	The reason code indicating the result of the put for this object. Only included if MQRR structures are provided and the reason code for the MQPUT is MQRC_MULTIPLE_REASONS
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	2
Type:	MQCFIN

MsgId

Description:	Message identifier. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_MSG_ID
PCF Parameter:	MQBACF_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

CorrelId

Description:	Correlation identifier. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_CORREL_ID
PCF Parameter:	MQBACF_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

GroupId

Description:	Group identifier. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_GROUP_ID
PCF Parameter:	MQBACF_GROUP_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_GROUP_ID_LENGTH

Feedback

Description:	Feedback. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_FEEDBACK
PCF Parameter:	MQIACF_FEEDBACK
Trace level:	2
Type:	MQCFIN

AccountingToken

Description:	AccountingToken. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_ACCOUNTING_TOKEN
PCF Parameter:	MQBACF_ACCOUNTING_TOKEN

Trace level:	2
Type:	MQCFBS
Length:	MQ_ACCOUNTING_TOKEN_LENGTH.

MQPUT1
Application has started the MQPUT1 MQI function

ObjectType

Description:	The object type passed in MQOT.ObjectType
PCF Parameter:	MQIACF_OBJECT_TYPE
Trace level:	1
Type:	MQCFIN

ObjectName

Description:	The name of the object passed to the MQI call before any queue name resolution is attempted.
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	1
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

ObjectQMgrName

Description:	The name of the object queue manager passed to the MQI call before any queue name resolution is attempted.
PCF Parameter:	MQCACF_OBJECT_Q_MGR_NAME
Trace level:	1
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

PutOptions

Description:	The put options from MQPMO.Options

PCF Parameter:	MQIACF_PUT_OPTIONS
Trace level:	1
Type:	MQCFIN

AlternateUserId

Description:	Only included if MQPMO_ALTERNATE_USER_AUTHORITY is specified.
PCF Parameter:	MQCACF_ALTERNATE_USERID
Trace level:	2
Type:	MQCFST
Length:	MQ_USER_ID_LENGTH

RecsPresent

Description:	The number of object name records present
PCF Parameter:	MQIACF_RECS_PRESENT
Trace level:	1
Type:	MQCFIN

KnownDestCount

Description:	Number of local queues opened successfully
PCF Parameter:	MQIACF_KNOWN_DEST_COUNT
Trace level:	1
Type:	MQCFIN

UnknownDestCount

Description:	Number of remote queues opened successfully
PCF Parameter:	MQIACF_UNKNOWN_DEST_COUNT
Trace level:	1
Type:	MQCFIN

InvalidDestCount

Description:	Number of queues that failed to open
PCF Parameter:	MQIACF_INVALID_DEST_COUNT
Trace level:	1
Type:	MQCFIN

MsgBuffer

Description:	Message data.
PCF Parameter:	MQBACF_MESSAGE_DATA
Trace level:	1
Type:	MQCFBS
Length:	Length is governed by the TRACEDATA() parameter set in the APPTRACE configuration. If TRACEDATA=NONE then this parameter is omitted.

MsgLength

Description:	Length of the message.
PCF Parameter:	MQIACF_MSG_LENGTH
Trace level:	1
Type:	MQCFIN

HighResTime

Description:	Time of operation in microseconds since midnight, January 1st 1970 (UTC)
	Note: The accuracy of this timer will vary according to platform support for high a resolution timer.
PCF Parameter:	MQIAMO64_HIGHRES_TIME
Trace level:	2
Type:	MQCFIN64

ResolvedQName

Description:	The name of the queue after queue name resolution has been performed.
PCF Parameter:	MQCACF_RESOLVED_Q_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

ResolvedQMgrName

Description:	The queue manager name after name resolution has been performed.
PCF Parameter:	MQCACF_RESOLVED_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH

ResolvedLocalQName[4]

Description:	Contains the local queue name after name resolution has been carried out
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type:	MQCFST

ResolvedLocalQMgrName[4]

Description:	Contains the local queue manager name after name resolution has been carried out.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_MGR_NAME_LENGTH

AlternateSecurityId

Description:	Alternate security identifier. Only present if MQOD.Version is equal or greater than MQOD_VERSION_3 and MQOD.AlternateSecurityId is not equal to MQSID_NONE.
PCF Parameter:	MQBACF_ALTERNATE_SECURITYID
Trace level:	2
Type:	MQCFBS
Length:	MQ_SECURITY_ID_LENGTH

ObjectString

Description:	Long object name. Only included if MQOD.Version is equal or greater than MQOD_VERSION_4 and the VSLength field of MQOD.ObjectString is MQVS_NULL_TERMINATED or greater than zero.
PCF Parameter:	MQCACF_OBJECT_STRING
Trace level:	2
Type:	MQCFST
Length:	Length varies.

ResObjectString

Description:	The long object name after the queue manager resolves the name provided in the ObjectName field. Only included for topics and queue aliases that reference a topic object if MQOD.Version is equal or greater than MQOD_VERSION_4 and VSLength is MQVS_NULL_TERMINATED or greater than zero.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type:	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the resolved (base) object being opened. Only included if MQOD.Version is equal or greater than MQOD_VERSION_4. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type:	MQCFIN

ReportOptions

Description:	Message report options
PCF Parameter:	MQIACF_REPORT
Trace level:	2
Type:	MQCFIN

MsgType

Description:	Type of message

PCF Parameter:	MQIACF_MSG_TYPE
Trace level:	2
Type:	MQCFIN

Expiry

Description:	Message lifetime
PCF Parameter:	MQIACF_EXPIRY
Trace level:	2
Type:	MQCFIN

Format

Description:	Format name of message data
PCF Parameter:	MQCACH_FORMAT_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_FORMAT_LENGTH

Priority

Description:	Message priority
PCF Parameter:	MQIACF_PRIORITY
Trace level:	2
Type:	MQCFIN

Persistence

Description:	Message persistence
PCF Parameter:	MQIACF_PERSISTENCE
Trace level:	2
Type:	MQCFIN

MsgId

Description:	Message identifier
PCF Parameter:	MQBACF_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

CorrelId

PCF Parameter:	Correlation identifier
Description:	MQBACF_CORREL_ID
Trace level:	2
Type:	MQCFBS

Length: MQ_CORREL_ID_LENGTH

ReplyToQueue

Description:

PCF Parameter: MQCACF_REPLY_TO_Q

Trace level: 2

Type: MQCFST

Length: MQ_Q_NAME_LENGTH

ReplyToQMgr

Description:

PCF Parameter: MQCACF_REPLY_TO_Q_MGR

Trace level: 2

Type: MQCFST

Length: MQCFST

CodedCharSetId

Description: Character set identifier of message data

PCF Parameter: MQIA_CODED_CHAR_SET_ID

Trace level: 2

Type: MQCFIN

Encoding

Description: Numeric encoding of message data.

PCF Parameter: MQIACF_ENCODING

Trace level: 2

Type: MQCFIN

PutDate

Description:

PCF Parameter: MQCACF_PUT_DATE

Trace level: 2

Type: MQCFST

Length: MQ_PUT_DATE_LENGTH

PutTime

Description:

PCF Parameter: MQCACF_PUT_TIME

Trace level: 2

Type: MQCFST

Length: MQ_PUT_TIME_LENGTH

PolicyName

Description:	The policy name that was applied to this message.
	Note: AMS protected messages only
PCF Parameter:	MQCA_POLICY_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_OBJECT_NAME_LENGTH

XmitqMsgId

Description:	The message ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

XmitqCorrelId

Description:	The correlation ID of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQBACF_XQH_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

XmitqPutTime

Description:	The put time of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_TIME
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_TIME_LENGTH

XmitqPutDate

Description:	The put date of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_PUT_DATE
Trace level:	2
Type:	MQCFST
Length:	MQ_PUT_DATE_LENGTH

XmitqRemoteQName

Description:	The remote queue destination of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

XmitqRemoteQMgr

Description:	The remote queue manager destination of the message in the transmission queue header.
	Note: Only when Format is MQFMT_XMIT_Q_HEADER
PCF Parameter:	MQCACF_XQH_REMOTE_Q_MGR
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

PutMsgOptsStructure

Description:	The MQPMO structure.
PCF Parameter:	MQBACF_MQPMO_STRUCT
Trace level:	3
Type:	MQCFBS
Length:	The length in bytes of the MQPMO structure (actual size depends on structure version)

MQPUT1 AppActivityDistList PCF Group Header Structure

If the MQPUT1 function is putting to a distribution list, then the variable parameters include one AppActivityDistList PCF group. For each of the queues in the distribution list, see "Application Activity Distribution List PCF Group Header Structure" on page 235. The AppActivityDistList PCF group combines information from the MQOR, MQPMR, and MQRR structures to identify the objects, and the PUT parameters , and indicate the result of the PUT operation on each queue. For MQPUT1 operations the AppActivityDistList group contains some or all of the following parameters (the CompCode, Reason, ObjectName, and ObjectQMgrName is present if the reason code is MQRC_MULTIPLE_REASONS and the other parameters is determined by the MQPMO.PutMsgRecFields field):

CompCode

Description:	The completion code indicating the result of the put for this object. Only included if MQRR structures are provided and the reason code for the MQPUT1 is MQRC_MULTIPLE_REASONS
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	2
Type:	MQCFIN

---

4 The ResolvedLocalQName parameter is only included if it is different from the ResolvedQName parameter.

Reason

Description:	The reason code indicating the result of the put for this object. Only included if MQRR structures are provided and the reason code for the MQPUT1 is MQRC_MULTIPLE_REASONS
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	2
Type:	MQCFIN

ObjectName

Description:	The name of a queue in the distribution list. Only included if MQOR structures are provided.
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	2
Type:	MQCFST
Length:	MQ_Q_NAME_LENGTH

MsgId

Description:	Message identifier. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_MSG_ID
PCF Parameter:	MQBACF_MSG_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_MSG_ID_LENGTH

CorrelId

Description:	Correlation identifier. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_CORREL_ID
PCF Parameter:	MQBACF_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

GroupId

Description:	Group identifier. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_GROUP_ID
PCF Parameter:	MQBACF_GROUP_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_GROUP_ID_LENGTH

Feedback

Description:	Feedback. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_FEEDBACK

PCF Parameter:	MQIACF_FEEDBACK
Trace level:	2
Type:	MQCFIN

AccountingToken

Description:	AccountingToken. Only included if MQPMR structures are provided.and PutMsgRecFields includes MQPMRF_ACCOUNTING_TOKEN
PCF Parameter:	MQBACF_ACCOUNTING_TOKEN
Trace level:	2
Type:	MQCFBS
Length:	MQ_ACCOUNTING_TOKEN_LENGTH.

MQSET

Application has started the MQSET MQI function

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type:	MQCFIN

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

SelectorCount

Description:	The count of selectors that are supplied in the Selectors array.
PCF Parameter:	MQIACF_SELECTOR_COUNT
Trace level:	2
Type:	MQCFIN

Selectors

Description:	The list of attributes (integer or character) whose values are being updated by MQSET.
PCF Parameter:	MQIACF_SELECTORS

Trace level:	2
Type:	MQCFIL

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH.

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type	MQCFIN

IntAttrCount

Description:	The number of integer attributes to be updated by the set operation.
PCF Parameter:	MQIACF_INTATTR_COUNT
Trace level:	3
Type:	MQCFIN

IntAttrs

Description:	The integer attribute values
PCF Parameter:	MQIACF_INT_ATTRS
Trace level:	3
Type:	MQCFIL
Range:	This parameter is only present if IntAttrCount is > 0

CharAttrs

Description:	The character attributes to be updated by the set operation. The values are concatenated together.
PCF Parameter:	MQCACF_CHAR_ATTRS

Trace level:	3
Type:	MQCFST
Range:	This parameter is only included if CharAttrLength is > 0

MQSUB
Application has started the MQSUB MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

SubHandle

Description:	The subscription handle
PCF Parameter:	MQIACF_HSUB
Trace level:	1
Type:	MQCFIN

ObjectHandle

Description:	The object handle
PCF Parameter:	MQIACF_HOBJ
Trace level:	1
Type:	MQCFIN

Options

Description:	Subscription options
PCF Parameter:	MQIACF_SUB_OPTIONS
Trace level:	1
Type:	MQCFIN

ObjectName

Description:	The name of the object.
PCF Parameter:	MQCACF_OBJECT_NAME
Trace level:	1
Type:	MQCFST

	Length:	MQ_Q_NAME_LENGTH

ObjectString

	Description:	Long object name.
	PCF Parameter:	MQCACF_OBJECT_STRING
	Trace level:	1
	Type:	MQCFST
	Range:	Only included if the VSLength field of MQSD.ObjectString is greater than zero or MQVS_NULL_TERMINATED.
	Length:	Length varies.

AlternateUserId

	Description:	
	PCF Parameter:	MQCACF_ALTERNATE_USERID
	Trace level:	2
	Type:	MQCFST
	Range:	Only included if MQSO_ALTERNATE_USER_AUTHORITY is specified.
	Length:	MQ_USER_ID_LENGTH

AlternateSecurityId

	Description:	Alternate security identifier.
	PCF Parameter:	MQBACF_ALTERNATE_SECURITYID
	Trace level:	2
	Type:	MQCFBS
	Range:	Only present if MQSO_ALTERNATE_USER_AUTHORITY is specified and MQSD.AlternateSecurityId is not equal to MQSID_NONE.
	Length:	MQ_SECURITY_ID_LENGTH

SubName

	Description:	Subscription Name
	PCF Parameter:	MQCACF_SUB_NAME
	Trace level:	2
	Type:	MQCFST
	Range:	Only included if the VSLength field of MQSD.SubName is greater than zero or MQVS_NULL_TERMINATED.
	Length:	Length varies.

SubUserData

	Description:	Subscription User Data
	PCF Parameter:	MQCACF_SUB_USER_DATA
	Trace level:	2
	Type:	MQCFST

Range:	Only included if the VSLength field of MQSD.SubName is greater than zero or MQVS_NULL_TERMINATED.
Length:	Length varies.

SubCorrelId

Description:	Subscription Correlation identifier
PCF Parameter:	MQBACF_SUB_CORREL_ID
Trace level:	2
Type:	MQCFBS
Length:	MQ_CORREL_ID_LENGTH

SelectionString

Description:	Selection string.
PCF Parameter:	MQCACF_SELECTION_STRING
Trace level:	2
Type:	MQCFST
Range:	Only included if the VSLength field of MQSD. SelectionString is MQVS_NULL_TERMINATED or greater than zero.
Length:	Length varies.

ResolvedQName

Description:	The queue name referred to by the ObjectHandle, when ResolvedType is MQOT_Q.
PCF Parameter:	MQCACF_RESOLVED_LOCAL_Q_NAME
Trace level:	2
Type	MQCFST
Length:	MQ_Q_NAME_LENGTH.

ResObjectString

Description:	The object name referred to by the ObjectHandle, when ResolvedType is MQOT_TOPIC.
PCF Parameter:	MQCACF_RESOLVED_OBJECT_STRING
Trace level:	2
Type	MQCFST
Length:	Length varies.

ResolvedType

Description:	The type of the object referred to by the ObjectHandle. Possible values are MQOT_Q, MQOT_TOPIC, or MQOT_NONE.
PCF Parameter:	MQIACF_RESOLVED_TYPE
Trace level:	2
Type	MQCFIN

SubDescriptorStructure

Description:	The MQSD structure.
PCF Parameter:	MQBACF_MQSD_STRUCT
Trace level:	3
Type:	MQCFBS
Length:	The length in bytes of the MQSD structure.

MQSUBRQ

Application has started the MQSUBRQ MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

SubHandle

Description:	The subscription handle
PCF Parameter:	MQIACF_HSUB
Trace level:	1
Type:	MQCFIN

SubOptions

Description:	The sub options from MQSB.Options
PCF Parameter:	MQIACF_SUBRQ_OPTIONS
Trace level:	2
Type:	MQCFIN

Action

Description:	The subscription request action (MQSR_*)
PCF Parameter:	MQIACF_SUBRQ_ACTION
Trace level:	2
Type:	MQCFIN

NumPubs

Description:	The number of publications sent as a result of this call (from MQSB.NumPubs)

PCF Parameter:	MQIACF_NUM_PUBS
Trace level:	2
Type:	MQCFIN

MQSTAT
Application has started the MQSTAT MQI function

CompCode

Description:	The completion code indicating the result of the operation
PCF Parameter:	MQIACF_COMP_CODE
Trace level:	1
Type:	MQCFIN

Reason

Description:	The reason code result of the operation
PCF Parameter:	MQIACF_REASON_CODE
Trace level:	1
Type:	MQCFIN

Type

Description:	Type of status information being requested
PCF Parameter:	MQIACF_STATUS_TYPE
Trace level:	2
Type:	MQCFIN

StatusStructure

Description:	The MQSTS structure.
PCF Parameter:	MQBACF_MQSTS_STRUCT
Trace level:	3
Type:	MQCFBS
Length:	The length in bytes of the MQSTS structure (actual size depends on structure version)

Variable Parameters for Application Activity XA Operations

XA operations are API calls that applications can make to enable MQ to participate in a transaction. The parameters for each operation are defined in the following section.

The trace level indicates the level of trace granularity that is required for the parameters to be included in the trace. The possible trace level values are:

1. Low

   The parameter is included when "low", "medium" or "high" activity tracing is configured for an application. This setting means that a parameter is always included in the AppActivityData group for the operation. This set of parameters is sufficient to trace the MQI calls an application makes, and to see if they are successful.

2. Medium

The parameter is only included in the `AppActivityData` group for the operation when "medium" or "high" activity tracing is configured for an application. This set of parameters adds information about the resources, for example, queue and topic names used by the application.

3. High

   The parameter is only included in the `AppActivityData` group for the operation when "high" activity tracing is configured for an application. This set of parameters includes memory dumps of the structures passed to the MQI and XA functions. For this reason, it contains more information about the parameters used in MQI and XA calls. The structure memory dumps are shallow copies of the structures. To avoid erroneous attempts to dereference pointers, the pointer values in the structures are set to `NULL`.

   Note: The version of the structure that is dumped is not necessarily identical to the version used by an application. The structure can be modified by an API crossing exit, by the activity trace code, or by the queue manager. A queue manager can modify a structure to a later version, but the queue manager never changes it to an earlier version of the structure. To do so, would risk losing data.

AXREG
Application has started the AXREG AX function

XID

Description:	The XID structure
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

AXUNREG
Application has started the AXUNREG AX function

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XACLOSE
Application has started the XACLOSE AX function

Xa_info

Description:	Information used to initialize the resource manager.
PCF Parameter:	MQCACF_XA_INFO
Trace level:	1
Type:	MQCFST

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code

PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XACOMMIT
Application has started the XACOMMIT AX function

XID

Description:	The XID structure
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XACOMPLETE
Application has started the XACOMPLETE AX function

Handle

Description:	Handle to async operation
PCF Parameter:	MQIACF_XA_HANDLE
Trace level:	1
Type:	MQCFIN

Retval

Description:	Return value of the asynchronous function
PCF Parameter:	MQIACF_XA_RETVAL

> Trace level: 1
>
> Type: MQCFINMQCFBS

Rmid

> Description: Resource manager identifier
>
> PCF Parameter: MQIACF_XA_RMID
>
> Trace level: 1
>
> Type: MQCFIN

Flags

> Description: Flags
>
> PCF Parameter: MQIACF_XA_FLAGS
>
> Trace level: 1
>
> Type: MQCFIN

XARetCode

> Description: Return code
>
> PCF Parameter: MQIACF_XA_RETCODE
>
> Trace level: 1
>
> Type: MQCFIN

XAEND

Application has started the XAEND AX function

XID

> Description: The XID structure
>
> PCF Parameter: MQBACF_XA_XID
>
> Trace level: 1
>
> Type: MQCFBS
>
> Length: Sizeof(XID)

Rmid

> Description: Resource manager identifier
>
> PCF Parameter: MQIACF_XA_RMID
>
> Trace level: 1
>
> Type: MQCFIN

Flags

> Description: Flags
>
> PCF Parameter: MQIACF_XA_FLAGS
>
> Trace level: 1
>
> Type: MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XAFORGET

Application has started the AXREG AX function

XID

Description:	The XID structure
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XAOPEN

Application has started the XAOPEN AX function

Xa_info

Description:	Information used to initialize the resource manager.
PCF Parameter:	MQCACF_XA_INFO
Trace level:	1
Type:	MQCFST

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XAPREPARE
Application has started the XAPREPARE AX function

XID

Description:	The XID structure
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code

PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XARECOVER

Application has started the XARECOVER AX function

Count

Description:	Count of XIDs
PCF Parameter:	MQIACF_XA_COUNT
Trace level:	1
Type:	MQCFIN

XIDs

Description:	The XID structures
	Note: There are multiple instances of this PCF parameter - one for every XID structure up to Count XIDs
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XAROLLBACK
Application has started the XAROLLBACK AX function

XID

Description:	The XID structure
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description:	Flags
PCF Parameter:	MQIACF_XA_FLAGS
Trace level:	1
Type:	MQCFIN

XARetCode

Description:	Return code
PCF Parameter:	MQIACF_XA_RETCODE
Trace level:	1
Type:	MQCFIN

XASTART
Application has started the XASTART AX function

XID

Description:	The XID structure
PCF Parameter:	MQBACF_XA_XID
Trace level:	1
Type:	MQCFBS
Length:	Sizeof(XID)

Rmid

Description:	Resource manager identifier
PCF Parameter:	MQIACF_XA_RMID
Trace level:	1
Type:	MQCFIN

Flags

Description: Flags

PCF Parameter: MQIACF_XA_FLAGS

Trace level: 1

Type: MQCFIN

XARetCode

Description: Return code

PCF Parameter: MQIACF_XA_RETCODE

Trace level: 1

Type: MQCFIN

Real-time monitoring

Real-time monitoring is a technique that allows you to determine the current state of queues and channels within a queue manager. The information returned is accurate at the moment the command was issued.

A number of commands are available that when issued return real-time information about queues and channels. Information can be returned for one or more queues or channels and can vary in quantity. Real-time monitoring can be used in the following tasks:

- Helping system administrators understand the steady state of their IBM WebSphere MQ system. This helps with problem diagnosis if a problem occurs in the system.
- Determining the condition of your queue manager at any moment, even if no specific event or problem has been detected.
- Assisting with determining the cause of a problem in your system.

With real-time monitoring, information can be returned for either queues or channels. The amount of real-time information returned is controlled by queue manager, queue, and channel attributes.

- You monitor a queue by issuing commands to ensure that the queue is being serviced properly. Before you can use some of the queue attributes, you must enable them for real-time monitoring.
- You monitor a channel by issuing commands to ensure that the channel is running properly. Before you can use some of the channel attributes, you must enable them for real-time monitoring.

Real-time monitoring for queues and channels is in addition to, and separate from, performance and channel event monitoring.

Attributes that control real-time monitoring

Some queue and channel status attributes hold monitoring information, if real-time monitoring is enabled. If real-time monitoring is not enabled, no monitoring information is held in these monitoring attributes. Examples demonstrate how you can use these queue and channel status attributes.

You can enable or disable real-time monitoring for individual queues or channels, or for multiple queues or channels. To control individual queues or channels, set the queue attribute MONQ or the channel attribute MONCHL, to enable or disable real-time monitoring. To control many queues or channels together, enable or disable real-time monitoring at the queue manager level by using the queue manager attributes MONQ and MONCHL. For all queue and channel objects with a monitoring attribute that is specified with the default value, QMGR, real-time monitoring is controlled at the queue manager level.

Automatically defined cluster-sender channels are not WebSphere MQ objects, so do not have attributes in the same way as channel objects. To control automatically defined cluster-sender channels, use the

queue manager attribute, MONACLS. This attribute determines whether automatically defined cluster-sender channels within a queue manager are enabled or disabled for channel monitoring.

For real-time monitoring of channels, you can set the MONCHL attribute to one of the three monitoring levels: low, medium, or high. You can set the monitoring level either at the object level or at the queue manager level. The choice of level is dependent on your system. Collecting monitoring data might require some instructions that are relatively expensive computationally, such as obtaining system time. To reduce the effect of real-time monitoring, the medium and low monitoring options measure a sample of the data at regular intervals rather than collecting data all the time. summarizes the monitoring levels available for real-time monitoring of channels:

Table 29. Monitoring levels		
Level	**Description**	**Usage**
Low	Measure a small sample of the data, at regular intervals.	For objects that process a high volume of messages.
Medium	Measure a sample of the data, at regular intervals.	For most objects.
High	Measure all data, at regular intervals.	For objects that process only a few messages per second, on which the most current information is important.

For real-time monitoring of queues, you can set the MONQ attribute to one of the three monitoring levels, low, medium or high. However, there is no distinction between these values. The values all enable data collection, but do not affect the size of the sample.

Examples

The following examples demonstrate how to set the necessary queue, channel, and queue manager attributes to control the level of monitoring. For all of the examples, when monitoring is enabled, queue and channel objects have a medium level of monitoring.

1. To enable both queue and channel monitoring for all queues and channels at the queue manager level, use the following commands:

```
ALTER QMGR MONQ(MEDIUM) MONCHL(MEDIUM)
ALTER QL(Q1) MONQ(QMGR)
ALTER CHL(QM1.TO.QM2) CHLTYPE(SDR) MONCHL(QMGR)
```

2. To enable monitoring for all queues and channels, with the exception of local queue, Q1, and sender channel, QM1.TO.QM2, use the following commands:

```
ALTER QMGR MONQ(MEDIUM) MONCHL(MEDIUM)
ALTER QL(Q1) MONQ(OFF)
ALTER CHL(QM1.TO.QM2) CHLTYPE(SDR) MONCHL(OFF)
```

3. To disable both queue and channel monitoring for all queues and channels, with the exception of local queue, Q1, and sender channel, QM1.TO.QM2, use the following commands:

```
ALTER QMGR MONQ(OFF) MONCHL(OFF)
ALTER QL(Q1) MONQ(MEDIUM)
ALTER CHL(QM1.TO.QM2) CHLTYPE(SDR) MONCHL(MEDIUM)
```

4. To disable both queue and channel monitoring for all queues and channels, regardless of individual object attributes, use the following command:

```
ALTER QMGR MONQ(NONE) MONCHL(NONE)
```

5. To control the monitoring capabilities of automatically defined cluster-sender channels use the following command:

```
ALTER QMGR MONACLS(MEDIUM)
```

6. To specify that automatically defined cluster-sender channels are to use the queue manager setting for channel monitoring, use the following command:

```
ALTER QMGR MONACLS(QMGR)
```

Related concepts

"Real-time monitoring" on page 269
Real-time monitoring is a technique that allows you to determine the current state of queues and channels within a queue manager. The information returned is accurate at the moment the command was issued.

Working with queue managers

Related tasks

"Displaying queue and channel monitoring data" on page 271
To display real-time monitoring information for a queue or channel, use either the IBM WebSphere MQ Explorer or the appropriate MQSC command. Some monitoring fields display a comma-separated pair of indicator values, which help you to monitor the operation of your queue manager. Examples demonstrate how you can display monitoring data.

Monitoring (MONCHL)

Displaying queue and channel monitoring data

To display real-time monitoring information for a queue or channel, use either the IBM WebSphere MQ Explorer or the appropriate MQSC command. Some monitoring fields display a comma-separated pair of indicator values, which help you to monitor the operation of your queue manager. Examples demonstrate how you can display monitoring data.

About this task

Monitoring fields that display a pair of values separated by a comma provide short term and long term indicators for the time measured since monitoring was enabled for the object, or from when the queue manager was started:

- The short term indicator is the first value in the pair and is calculated in a way such that more recent measurements are given a higher weighting and will have a greater effect on this value. This gives an indication of recent trend in measurements taken.

- The long term indicator in the second value in the pair and is calculated in a way such that more recent measurements are not given such a high weighting. This gives an indication of the longer term activity on performance of a resource.

These indicator values are most useful to detect changes in the operation of your queue manager. This requires knowledge of the times these indicators show when in normal use, in order to detect increases in these times. By collecting and checking these values regularly you can detect fluctuations in the operation of your queue manager. This can indicate a change in performance.

Obtain real-time monitoring information as follows:

Procedure

1. To display real-time monitoring information for a queue, use either the IBM WebSphere MQ Explorer or the MQSC command DISPLAY QSTATUS, specifying the optional parameter MONITOR.

2. To display real-time monitoring information for a channel, use either the IBM WebSphere MQ Explorer or the MQSC command DISPLAY CHSTATUS, specifying the optional parameter MONITOR.

Example

The queue, Q1, has the attribute MONQ set to the default value, QMGR, and the queue manager that owns the queue has the attribute MONQ set to MEDIUM. To display the monitoring fields collected for this queue, use the following command:

```
DISPLAY QSTATUS(Q1) MONITOR
```

The monitoring fields and monitoring level of queue, Q1 are displayed as follows:

```
QSTATUS(Q1)
TYPE(QUEUE)
MONQ(MEDIUM)
QTIME(11892157,24052785)
MSGAGE(37)
LPUTDATE(2005-03-02)
LPUTTIME(09.52.13)
LGETDATE(2005-03-02)
LGETTIME(09.51.02)
```

The sender channel, QM1.TO.QM2, has the attribute MONCHL set to the default value, QMGR, and the queue manager that owns the queue has the attribute MONCHL set to MEDIUM. To display the monitoring fields collected for this sender channel, use the following command:

```
DISPLAY CHSTATUS(QM1.TO.QM2) MONITOR
```

The monitoring fields and monitoring level of sender channel, QM1.TO.QM2 are displayed as follows:

```
CHSTATUS(QM1.TO.QM2)
XMITQ(Q1)
CONNAME(127.0.0.1)
CURRENT
CHLTYPE(SDR)
STATUS(RUNNING)
SUBSTATE(MQGET)
MONCHL(MEDIUM)
XQTIME(755394737,755199260)
NETTIME(13372,13372)
EXITTIME(0,0)
XBATCHSZ(50,50)
COMPTIME(0,0)
STOPREQ(NO)
RQMNAME(QM2)
```

Related concepts
"Real-time monitoring" on page 269
Real-time monitoring is a technique that allows you to determine the current state of queues and channels within a queue manager. The information returned is accurate at the moment the command was issued.

Related reference
DISPLAY QSTATUS

Monitoring queues

Use this page to view tasks that help you to resolve a problem with a queue and the application that services that queue. Various monitoring options are available to determine the problem

Frequently, the first sign of a problem with a queue that is being serviced is that the number of messages on the queue (CURDEPTH) increases. If you expect an increase at certain times of day or under certain

workloads, an increasing number of messages might not indicate a problem. However, if you have no explanation for the increasing number of messages, you might want to investigate the cause.

You might have an application queue where there is a problem with the application, or a transmission queue where there is a problem with the channel. Additional monitoring options are available when the application that services the queue is a channel.

The following examples investigate problems with a particular queue, called Q1, and describe the fields that you look at in the output of various commands:

Determining whether your application has the queue open

If you have a problem with a queue, check whether your application has the queue open

About this task
Perform the following steps to determine whether your application has the queue open:

Procedure

1. Ensure that the application that is running against the queue is the application that you expect. Issue the following command for the queue in question:

   ```
   DISPLAY QSTATUS(Q1) TYPE(HANDLE) ALL
   ```

   In the output, look at the APPLTAG field, and check that the name of your application is shown. If the name of your application is not shown, or if there is no output at all, start your application.

2. If the queue is a transmission queue, look in the output at the CHANNEL field.

   If the channel name is not shown in the CHANNEL field, determine whether the channel is running.

3. Ensure that the application that is running against the queue has the queue open for input. Issue the following command:

   ```
   DISPLAY QSTATUS(Q1) TYPE(QUEUE) ALL
   ```

   In the output, look at the IPPROCS field to see if any application has the queue open for input. If the value is 0 and this is a user application queue, make sure that the application opens the queue for input to get the messages off the queue.

Checking that messages on the queue are available

If you have a large number of messages on the queue and your application is not processing any of those messages, check whether the messages on the queue are available to your application

About this task
Perform the following steps to investigate why your application is not processing messages from the queue:

Procedure

1. Ensure that your application is not asking for a specific message ID or correlation ID when it should be processing all the messages on the queue.

2. Although the current depth of the queue might show that there is an increasing number of messages on the queue, some messages on the queue might not be available to be got by an application, because they are not committed; the current depth includes the number of uncommitted MQPUTs of messages to the queue. Issue the following command:

   ```
   DISPLAY QSTATUS(Q1) TYPE(QUEUE) ALL
   ```

In the output, look at the UNCOM field to see whether there are any uncommitted messages on the queue.

3. If your application is attempting to get any messages from the queue, check whether the putting application is committing the messages correctly. Issue the following command to find out the names of applications that are putting messages to this queue:

```
DISPLAY QSTATUS(Q1) TYPE(HANDLE) OPENTYPE(OUTPUT)
```

4. Then issue the following command, inserting in <appltag> the APPLTAG value from the output of the previous command:

```
DISPLAY CONN(*) WHERE(APPLTAG EQ <appltag>) UOWSTDA UOWSTTI
```

This shows when the unit of work was started and will help you discover whether the application is creating a long running unit of work. If the putting application is a channel, you might want to investigate why a batch is taking a long time to complete.

Checking whether your application is getting messages off the queue

If you have a problem with a queue and the application that services that queue, check whether your application is getting messages off the queue

About this task
To check whether your application is getting messages off the queue, perform the following checks:

Procedure

1. Ensure that the application that is running against the queue is actually processing messages from the queue. Issue the following command:

```
DISPLAY QSTATUS(Q1) TYPE(QUEUE) ALL
```

In the output, look at the LGETDATE and LGETTIME fields which show when the last get was done from the queue.

2. If the last get from this queue was longer ago than expected, ensure that the application is processing messages correctly.

   If the application is a channel, check whether messages are moving through that channel

Determining whether the application can process messages fast enough

If messages are building up on the queue, but your other checks have not found any processing problems, check that the application can process messages fast enough. If the application is a channel, check that the channel can process messages fast enough.

About this task
To determine whether the application is processing messages fast enough, perform the following tests:

Procedure

1. Issue the following command periodically to gather performance data about the queue:

```
DISPLAY QSTATUS(Q1) TYPE(QUEUE) ALL
```

If the values in the QTIME indicators are high, or are increasing over the period, and you have already ruled out the possibility of long running Units of Work by checking that messages on the queue are available, the getting application might not be keeping up with the putting applications.

2. If your getting application cannot keep up with the putting applications, consider adding another getting application to process the queue.

   Whether you can add another getting application depends on the design of the application and whether the queue can be shared by more than one application. Features such as message grouping or getting by correlation ID might help to ensure that two applications can process a queue simultaneously.

Checking the queue when the current depth is not increasing

Even if the current depth of your queue is not increasing, it might still be useful to monitor the queue to check whether your application is processing messages correctly.

About this task
To gather performance data about the queue: Issue the following command periodically:

Procedure

Issue the following command periodically:

```
DISPLAY QSTATUS(Q1) TYPE(QUEUE) MSGAGE QTIME
```

In the output, if the value in MSGAGE increases over the period of time, and your application is designed to process all messages, this might indicate that some messages are not being processed at all.

Monitoring channels

Use this page to view tasks that help you to resolve a problem with a transmission queue and the channel that services that queue. Various channel monitoring options are available to determine the problem.

Frequently, the first sign of a problem with a queue that is being serviced is that the number of messages on the queue (CURDEPTH) increases. If you expect an increase at certain times of day or under certain workloads, an increasing number of messages might not indicate a problem. However, if you have no explanation for the increasing number of messages, you might want to investigate the cause.

You might have a problem with the channel that services a transmission queue. Various channel monitoring options are available to help you to determine the problem.

The following examples investigate problems with a transmission queue called QM2 and a channel called QM1.TO.QM2. This channel is used to send messages from queue manager, QM1, to queue manager, QM2. The channel definition at queue manager QM1 is either a sender or server channel, and the channel definition at queue manager, QM2, is either a receiver or requester channel.

Determining whether the channel is running

If you have a problem with a transmission queue, check whether the channel is running.

About this task
Perform the following steps to check the status of the channel that is servicing the transmission queue:

Procedure

1. Issue the following command to find out which channel you expect to process the transmission queue QM2:

```
DIS CHANNEL(*) WHERE(XMITQ EQ QM2)
```

In this example, the output of this command shows that the channel servicing the transmission queue is QM1.TO.QM2

2. Issue the following command to determine the status of the channel, QM1.TO.QM2:

```
DIS CHSTATUS(QM1.TO.QM2) ALL
```

3. Inspect the STATUS field of the output from the **CHSTATUS** command:
   - If the value of the STATUS field is RUNNING, check that the channel is moving messages
   - If the output from the command shows no status, or the value of the STATUS field is STOPPED, RETRY, BINDING, or REQUESTING, perform the appropriate step, as follows:

4. Optional: If the value of the STATUS field shows no status, the channel is inactive, so perform the following steps:

   a) If the channel should have been started automatically by a trigger, check that the messages on the transmission queue are available.

      If there are messages available on the transmission queue, check that the trigger settings on the transmission queue are correct.

   b) Issue the following command to start the channel again manually:

```
START CHANNEL(QM1.TO.QM2)
```

5. Optional: If the value of the STATUS field is STOPPED, perform the following steps:

   a) Check the error logs to determine why the channel stopped. If the channel stopped owing to an error, correct the problem.

      Ensure also that the channel has values specified for the retry attributes: *SHORTRTY* and *LONGRTY*. In the event of transient failures such as network errors, the channel will then attempt to restart automatically.

   b) Issue the following command to start the channel again manually:

```
START CHANNEL(QM1.TO.QM2)
```

6. Optional: If the value of the STATUS field is RETRY, perform the following steps:

   a) Check the error logs to identify the error, then correct the problem.

   b) Issue the following command to start the channel again manually:

```
START CHANNEL(QM1.TO.QM2)
```

   or wait for the channel to connect successfully on its next retry.

7. Optional: If the value of the STATUS field is BINDING or REQUESTING, the channel has not yet successfully connected to the partner. Perform the following steps:

   a) Issue the following command, at both ends of the channel, to determine the substate of the channel:

```
DIS CHSTATUS(QM1.TO.QM2) ALL
```

   Note:

   i) In some cases there might be a substate at one end of the channel only.

   ii) Many substates are transitory, so issue the command a few times to detect whether a channel is stuck in a particular substate.

   b) Check to determine what action to take:

Table 30. Substates seen with status binding or requesting

Initiating MCA substate [1]	Responding MCA substate [2]	Notes
NAMESERVER		The initiating MCA is waiting for a name server request to complete. Ensure that the correct host name has been specified in the channel attribute, CONNAME, and that your name servers are set up correctly.
SCYEXIT	SCYEXIT	The MCAs are currently *in conversation* through a security exit. For more information, see "Determining whether the channel can process messages fast enough" on page 279.
	CHADEXIT	The channel autodefinition exit is currently executing. For more information, see "Determining whether the channel can process messages fast enough" on page 279.
RCVEXIT SENDEXIT MSGEXIT MREXIT	RCVEXIT SENDEXIT MSGEXIT MREXIT	Exits are called at channel startup for MQXR_INIT. Review the processing in this part of your exit if this takes a long time. For more information, see "Determining whether the channel can process messages fast enough" on page 279.
SERIALIZE	SERIALIZE	This substate only applies to channels with a disposition of SHARED.
NETCONNECT		This substate is shown if there is a delay in connecting due to incorrect network configuration.
SSLHANDSHAKE	SSLHANDSHAKE	An SSL handshake consists of a number of sends and receives. If network times are slow, or connection to lookup CRLs are slow, this affects the time taken to do the handshake.

Notes:

i) The initiating MCA is the end of the channel which started the conversation. This can be senders, cluster-senders, fully-qualified servers and requesters. In a server-requester pair, it is the end from which you started the channel.

ii) The responding MCA is the end of the channel which responded to the request to start the conversation. This can be receivers, cluster-receivers, requesters (when the server or sender is started), servers (when the requester is started) and senders (in a requester-sender call-back pair of channels).

Checking that the channel is moving messages

If you have a problem with a transmission queue, check that the channel is moving messages

Before you begin

Issue the command DIS CHSTATUS(QM1.TO.QM2) ALL. If the value of the STATUS field is RUNNING, the channel has successfully connected to the partner system.

Check that there are no uncommitted messages on the transmission queue, as described in "Checking that messages on the queue are available" on page 273.

About this task

If there are messages available for the channel to get and send, perform the following checks:

Procedure

1. In the output from the display channel status command, DIS CHSTATUS(QM1.TO.QM2) ALL, look at the following fields:

    MSGS
    Number of messages sent or received (or, for server-connection channels, the number of MQI calls handled) during this session (since the channel was started).

    BUFSSENT
    Number of transmission buffers sent. This includes transmissions to send control information only.

    BYTSSENT
    Number of bytes sent during this session (since the channel was started). This includes control information sent by the message channel agent.

    LSTMSGDA
    Date when the last message was sent or MQI call was handled, see LSTMSGTI.

    LSTMSGTI
    Time when the last message was sent or MQI call was handled. For a sender or server, this is the time the last message (the last part of it if it was split) was sent. For a requester or receiver, it is the time the last message was put to its target queue. For a server-connection channel, it is the time when the last MQI call completed.

    CURMSGS
    For a sending channel, this is the number of messages that have been sent in the current batch. For a receiving channel, it is the number of messages that have been received in the current batch. The value is reset to zero, for both sending and receiving channels, when the batch is committed.

2. Determine whether the channel has sent any messages since it started. If any have been sent, determine when the last message was sent.

3. If the channel has started a batch that has not yet completed, as indicated by a non-zero value in CURMSGS, the channel might be waiting for the other end of the channel to acknowledge the batch. Look at the SUBSTATE field in the output and refer to :

Table 31. Sender and receiver MCA substates

Sender SUBSTATE	Receiver SUBSTATE	Notes
MQGET	RECEIVE	Normal states of a channel at rest.
SEND	RECEIVE	SEND is usually a transitory state. If SEND is seen it indicates that the communication protocol buffers have filled. This can indicate a network problem.
RECEIVE		If the sender is seen in RECEIVE substate for any length of time, it is waiting on a response, either to a batch completion or a heartbeat. You might want to check why a batch takes a long time to complete.

Note: You might also want to determine whether the channel can process messages fast enough, especially if the channel has a substate associated with exit processing.

Checking why a batch takes a long time to complete

Use this page to view some reasons why a batch can take a long time to complete.

About this task

When a sender channel has sent a batch of messages it waits for confirmation of that batch from the receiver, unless the channel is pipelined. The following factors can affect how long the sender channel waits:

Procedure

- Check whether the network is slow.

  A slow network can affect the time it takes to complete a batch. The measurements that result in the indicators for the NETTIME field are measured at the end of a batch. However, the first batch affected by a slowdown in the network is not indicated with a change in the NETTIME value because it is measured at the end of the batch.

- Check whether the channel is using message retry.

  If the receiver channel fails to put a message to a target queue, it might use message retry processing, rather than put the message to a dead-letter queue immediately. Retry processing can cause the batch to slow down. In between MQPUT attempts, the channel will have STATUS(PAUSED), indicating that it is waiting for the message retry interval to pass.

Determining whether the channel can process messages fast enough

If there messages are building up on the transmission queue, but you have found no processing problems, determine whether the channel can process messages fast enough.

Before you begin

Issue the following command repeatedly over a period of time to gather performance data about the channel:

```
DIS CHSTATUS(QM1.TO.QM2) ALL
```

About this task

Confirm that there are no uncommitted messages on the transmission queue, as described in "Checking that messages on the queue are available" on page 273, then check the XQTIME field in the output from the display channel status command. When the values of the XQTIME indicators are consistently high, or increase over the measurement period, the indication is that the channel is not keeping pace with the putting applications.

Perform the following tests:

Procedure

1. Check whether exits are processing.

   If exits are used on the channel that is delivering these messages, they might add to the time spent processing messages. To identify if this is the case, do the following checks:

   a) In the output of the command DIS CHSTATUS(QM1.TO.QM2) ALL, check the EXITTIME field.

      If the time spent in exits is higher than expected, review the processing in your exits for any unnecessary loops or extra processing, especially in message, send, and receive exits. Such processing affects all messages moved across the channel.

   b) In the output of the command DIS CHSTATUS(QM1.TO.QM2) ALL, check the SUBSTATE field.

      If the channel has of one of the following substates for a significant time, review the processing in your exits:

      - SCYEXIT
      - RCVEXIT
      - SENDEXIT
      - MSGEXIT
      - MREXIT

2. Check whether the network is slow.

If messages are not moving fast enough across a channel, it might be because the network is slow. To identify if this is the case, do the following checks:

a) In the output of the command DIS CHSTATUS(QM1.TO.QM2) ALL, check the NETTIME field.

These indicators are measured when the sending channel asks its partner for a response. This happens at the end of each batch and, when a channel is idle during heartbeating.

b) If this indicator shows that round trips are taking longer than expected, use other network monitoring tools to investigate the performance of your network.

3. Check whether the channel is using compression.

If the channel is using compression, this adds to the time spent processing messages. If the channel is using only one compression algorithm, do the following checks:

a) In the output of the command DIS CHSTATUS(QM1.TO.QM2) ALL, check the COMPTIME field.

These indicators show the time spent during compression or decompression.

b) If the chosen compression is not reducing the amount of data to send by the expected amount, change the compression algorithm.

4. If the channel is using multiple compression algorithms, do the following checks:

a) In the output of the command DIS CHSTATUS(QM1.TO.QM2) ALL, check the COMPTIME, COMPHDR, and COMPMSG fields.

b) Change the compression algorithms specified on the channel definition, or consider writing a message exit to override the channel's choice of compression algorithm for particular messages if the rate of compression, or choice of algorithm, is not providing the required compression or performance.

Solving problems with cluster channels

If you have a build up of messages on the SYSTEM.CLUSTER.TRANSMIT.QUEUE queue, the first step in diagnosing the problem is discovering which channel, or channels, are having a problem delivering messages.

About this task

To discover which channel, or channels, using the SYSTEM.CLUSTER.TRANSMIT.QUEUE are having a problem delivering messages. Perform the following checks:

Procedure

1. Issue the following command:

```
DIS CHSTATUS(*) WHERE(XQMSGSA GT 1)
```

Note: If you have a busy cluster that has many messages moving, consider issuing this command with a higher number to eliminate the channels that have only a few messages available to deliver.

2. Look through the output for the channel, or channels, that have large values in the field XQMSGSA. Determine why the channel is not moving messages, or is not moving them fast enough. Use the tasks outlined in "Monitoring channels" on page 275 to diagnose the problems with the channels found to be causing the build up.

Monitoring transmission queue switching

It is important that you monitor the process of cluster-sender channels switching transmission queues so that the impact on your enterprise is minimized. For example, you should not attempt this process when the workload is high or by switching many channels simultaneously.

The process of switching channels

The process used to switch channels is:

1. The channel opens the new transmission queue for input and starts getting messages from it (using get by correlation ID)
2. A background process is initiated by the queue manager to move any messages queued for the channel from its old transmission queue to its new transmission queue. While messages are being moved any new messages for the channel are queued to the old transmission queue to preserve sequencing. This process might take a while to complete if there are a large number of messages for the channel on its old transmission queue, or new messages are rapidly arriving.
3. When no committed or uncommitted messages remain queued for the channel on its old transmission queue then the switch is completed. New messages are now put directly to the new transmission queue.

To avoid the eventuality of numerous channels switching simultaneously IBM WebSphere MQ provides the ability to switch the transmission queue of one or more channels that are not running by using the **runswchl** command.

Monitoring the status of switch operations

To understand the status of switch operations administrators can perform the following actions:

- Monitor the queue manager error log (AMQERR01.LOG) where messages are output to indicate the following stages during the operation:
  - The switch operation has started
  - The moving of messages has started
  - Periodic updates on how many messages are left to move (if the switch operation does not complete quickly)
  - The moving of messages has completed
  - The switch operation has completed

  .
- Use the DISPLAY CLUSQMGR command to query the transmission queue that each cluster-sender channel is currently using.
- Run the **runswchl** command in query mode to ascertain the switching status of one or more channels. The output of this command identifies the following for each channel:
  - Whether the channel has a switch operation pending
  - Which transmission queue the channel is switching from and to
  - How many messages remain on the old transmission queue

  Each command is really useful, because in one invocation you can determine the status of every channel, the impact a configuration change has had and whether all switch operations have completed.

Potential issues that might occur

See Potential issues when switching transmission queues for a list of some issues that might be encountered when switching transmission queue, their causes, and most likely solutions.

The Windows performance monitor

In WebSphere MQ Version 7.0 and earlier versions, it was possible to monitor the performance of local queues on Windows systems by using the Windows performance monitor. As of WebSphere MQ Version 7.1, this method of performance monitoring is no longer available.

You can monitor queues on all supported platforms by using methods described in "Real-time monitoring" on page 269.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

This book contains information on intended programming interfaces that allow the customer to write programs to obtain the services of IBM 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

IBM, the IBM logo, ibm.com®, are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at "Copyright and trademark information"www.ibm.com/legal/copytrade.shtml. Other product and service names might be trademarks of IBM or other companies.

Microsoft™ and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

This product includes software developed by the Eclipse Project (http://www.eclipse.org/).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

IBM ®

Part Number: