MQSeries®

Event Monitoring

SC34-5760-01

Note!

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

Second edition (November 2000)

This edition applies to the following products:

- MQSeries for AIX®, Version 5 Release 1
- MQSeries for AS/400®, Version 5 Release 1
- MQSeries for AT&T GIS UNIX®, Version 2 Release 2
- MQSeries for Compaq OpenVMS, Version 2 Release 2.1.1
- MQSeries for Compaq Tru64 UNIX, Version 5 Release 1
- MQSeries for HP-UX, Version 5 Release 1
- MQSeries for OS/2® Warp, Version 5 Release 1
- MQSeries for OS/390®, Version 5 Release 2
- MQSeries for SINIX and DC/OSx, Version 2 Release 2
- MQSeries for Sun Solaris, Version 5 Release 1
- MQSeries for Sun Solaris, Intel® Platform Edition, Version 5 Release 1
- MQSeries for Tandem NonStop Kernel, Version 2 Release 2.0.1
- MQSeries for Windows NT®, Version 5 Release 1
- MQSeries for Windows®, Version 2 Release 1

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

Contents

Figures

Tables

About this book

This book describes the facilities available on MQSeries products for monitoring instrumentation events in a network of connected systems that use IBM® MQSeries products in different operating system environments.

MQSeries event monitoring is supported by the following MQSeries products:

- MQSeries for AS/400
- MQSeries for Compaq (DIGITAL) OpenVMS
- MQSeries for OS/2 Warp
- MQSeries for OS/390
- MQSeries for Tandem NonStop Kernel
- MQSeries for UNIX systems (**see Note below**)
- MQSeries for Windows NT
- MQSeries for Windows (V2.1 only)

Note: In this book, references to MQSeries for "UNIX systems" apply to:
IBM MQSeries for AIX
IBM MQSeries for AT&T GIS UNIX [1]
IBM MQSeries for Compaq Tru64 UNIX
IBM MQSeries for HP-UX
IBM MQSeries for SINIX and DC/OSx
IBM MQSeries for Sun Solaris (SPARC and Intel Platform Editions)

Who this book is for

Primarily, this book is intended for system programmers who write programs to monitor and administer MQSeries products.

What you need to know to understand this book

You should have:

- Experience in writing systems management applications
- An understanding of the Message Queue Interface (MQI)
- Experience of MQSeries programs in general, or familiarity with the content of the other books in the MQSeries library

How to use this book

This book is based on the Event Monitoring part of the *MQSeries Programmable System Management* book, SC33–1482–08.

- The introductory chapter of this book gives a general overview of instrumentation events.
- "Chapter 2. Understanding performance events" on page 15 goes into greater depth about performance events, specifically enabling and disabling them.
- "Chapter 3. Event message reference" on page 33 provides detailed reference information for specific events.

1. This platform has become NCR UNIX SVR4 MP-RAS, R3.0

- An example of using events is contained in "Chapter 4. Example of using instrumentation events" on page 111.
- There is a glossary and bibliography at the back of the book.

Chapter 1. An introduction to instrumentation events

This chapter discusses:
- “What instrumentation events are”
- “Types of event” on page 2
- “Enabling and disabling events” on page 8
- “Conditions that cause events” on page 10
- “Event queues” on page 11
- “Format of event messages” on page 12
- “Using event monitoring in an MQSeries network” on page 12
- “Monitoring performance on Windows NT” on page 13

What instrumentation events are

In MQSeries, an instrumentation event is a logical combination of conditions 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.

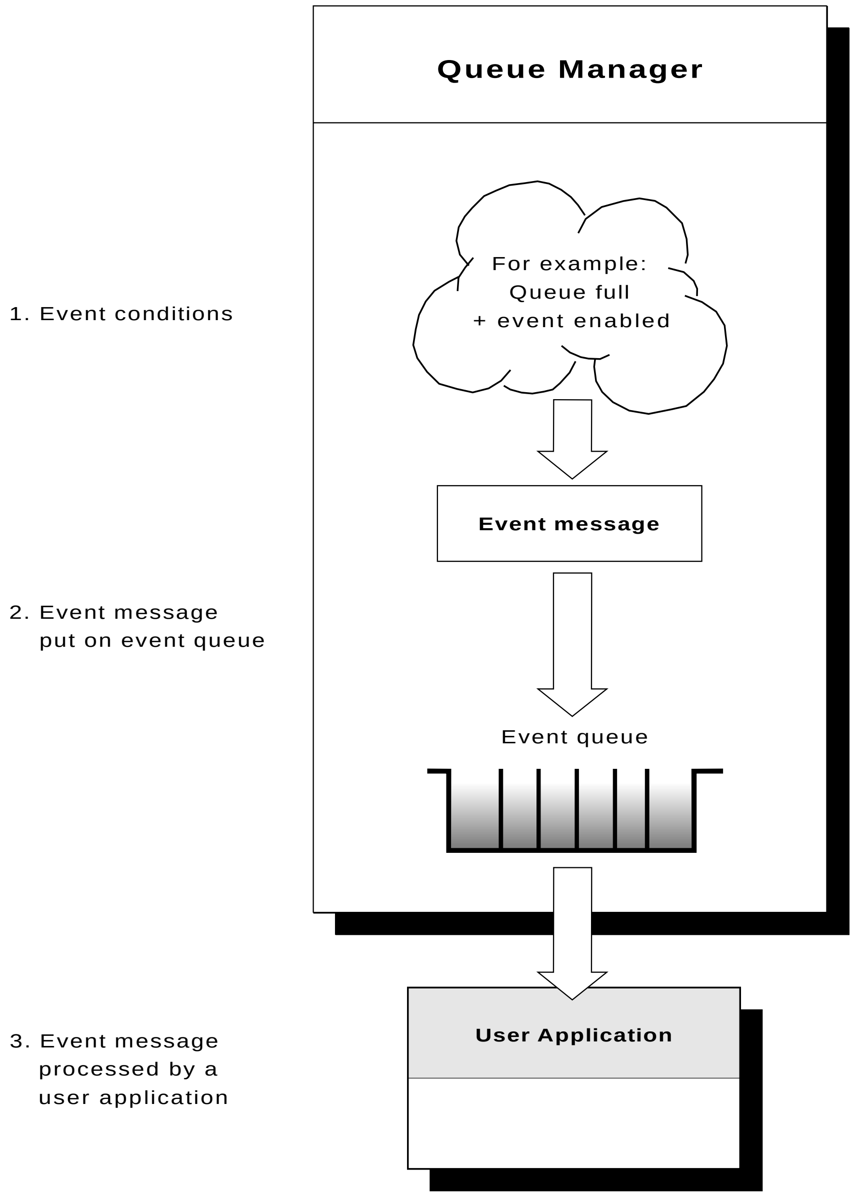
MQSeries instrumentation events provide information about errors, warnings, and other significant occurrences in a queue manager. You can, therefore, use these events to monitor the operation of queue managers (in conjunction with other methods such as NetView®). This chapter tells you what these events are, and how you use them.

Instrumentation events are supported by:
- MQSeries for AIX
- MQSeries for AS/400
- MQSeries for AT&T GIS UNIX
- MQSeries for Compaq OpenVMS
- MQSeries for Compaq Tru64 UNIX
- MQSeries for HP-UX
- MQSeries for OS/2 Warp
- MQSeries for OS/390
- MQSeries for SINIX and DC/OSx
- MQSeries for Sun Solaris
- MQSeries for Tandem NonStop Kernel
- MQSeries for Windows NT
- MQSeries for Windows V2.1

Note: This book does not discuss trigger events. When other MQSeries books discuss triggering, they sometimes refer to a *trigger event*. This occurs when a queue manager detects that the conditions for a trigger event have been met. For example, for a queue for which triggers are active, a message of the required priority has been put on a queue so that the trigger depth is reached. The result of a trigger event is that a trigger message is put onto an initiation queue and an application program is started. No other event messages as described in this book are involved (unless, for example, the initiation queue fills up and generates an instrumentation event).

Figure 1 on page 2 illustrates the concept of instrumentation events.

Event notification through event queues



Queue Manager
1. Event conditions
For example: Queue full + event enabled
Event message
2. Event message put on event queue
Event queue
3. Event message processed by a user application
User Application

Figure 1. Understanding instrumentation events

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:

- Gets the message from the queue.
- Processes the message to extract the event data. For an overview of event message formats, see “Format of event messages” on page 12. For detailed descriptions of the format of each event message, see “Event message format” on page 33.

Types of event

MQSeries instrumentation events may be categorized as follows:

Queue manager events
: These events are related to the definitions of resources within queue managers. For example, an application attempts to put a message to a queue that does not exist.

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

Channel events
: These events are reported by channels as a result of conditions detected during their operation. For example, when a channel instance is stopped.

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

Instrumentation events can be generated for queue managers running on Digital OpenVMS, OS/2, OS/390, OS/400®, Tandem NonStop Kernel, Windows 95, Windows 98, Windows NT, Windows 2000, and UNIX platforms. By incorporating these events into your own system management application, you can monitor the activities across many queue managers, across many different nodes, for multiple MQSeries applications. In particular, you can monitor all the nodes in your system from a single node (for those nodes that support MQSeries events) as shown in Figure 2 on page 4.

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

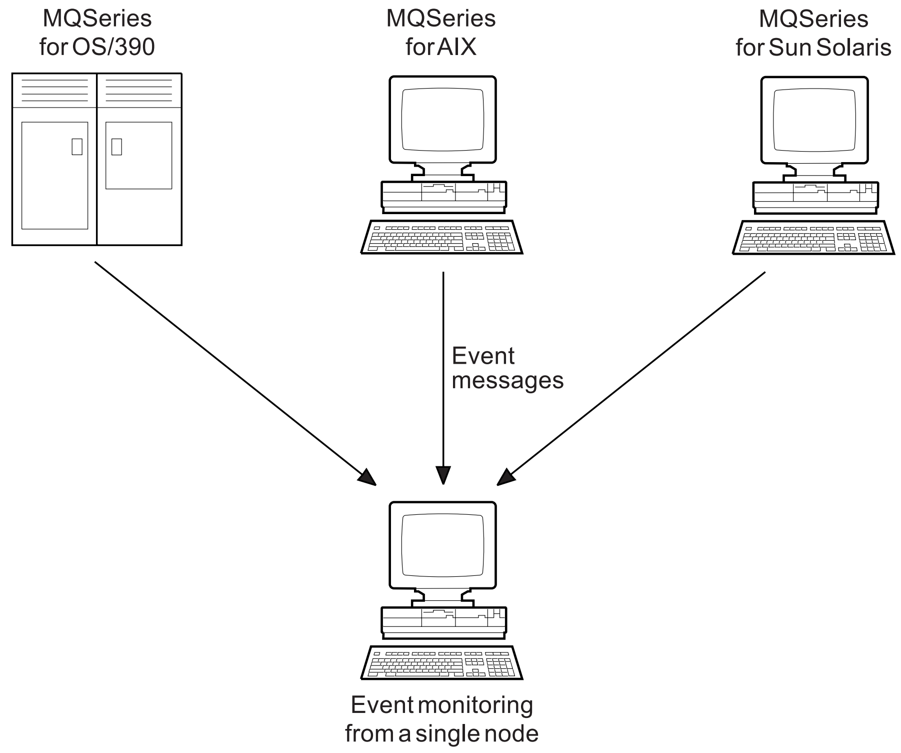


Figure 2. Monitoring queue managers across different platforms, on a single node

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

Queue manager events

These events are related to the definitions of resources within queue managers. For example, an application attempts to put a message to a queue that does not exist. The event messages for queue manager events are put on the SYSTEM.ADMIN.QMGR.EVENT queue. The following queue manager event types are supported:

- Authority (on OS/400, Windows NT, Compaq (DIGITAL) OpenVMS, Tandem NonStop Kernel, and UNIX systems only)
- Inhibit
- Local
- Remote
- Start and Stop (for OS/390: Start only)

For each event type in this list, there is a queue manager attribute that enables or disables the event type. See the *MQSeries MQSC Command Reference* for more information.

The conditions that give rise to the event (when enabled) include:

- 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.

  Note that a similar condition may occur during the internal operation of a queue manager, for example, when generating a report message. The reason code in an event message may match an MQI reason code, even though it is not associated with any application. Therefore you should 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 the processing of 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.

Authority events

Note

1. All authority events are valid on Digital OpenVMS, OS/400, Windows NT, and UNIX systems only.
2. Tandem NSK supports only Not Authorized (type 1).

Authority events indicate that an authorization violation has been detected. For example, an application attempts to open a queue for which it does not have the required authority, or a command is issued from a user ID that does not have the required authority.

For more information about the event data returned in authority event messages see:
“Not Authorized (type 1)” on page 72
“Not Authorized (type 2)” on page 74
“Not Authorized (type 3)” on page 76
“Not Authorized (type 4)” on page 78

Inhibit events

Inhibit events indicate that an MQPUT or MQGET operation has been attempted against a queue, where the queue is inhibited for puts or gets respectively.

For more information about the event data returned in inhibit event messages, see:
“Get Inhibited” on page 71
“Put Inhibited” on page 79

Local events

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, when an application attempts to access an object that has not been defined.

For more information about the event data returned in local event messages, see:
“Alias Base Queue Type Error” on page 46
“Unknown Alias Base Queue” on page 101
“Unknown Object Name” on page 105

Remote events

Remote events indicate that an application (or the queue manager) cannot access a (remote) queue on another queue manager. For example, when the transmission queue to be used is not correctly defined.

For more information about the event data returned in the remote event messages, see:
“Default Transmission Queue Type Error” on page 67
“Default Transmission Queue Usage Error” on page 69

“Queue Type Error” on page 93
“Remote Queue Name Error” on page 95
“Transmission Queue Type Error” on page 97
“Transmission Queue Usage Error” on page 99
“Unknown Default Transmission Queue” on page 103
“Unknown Remote Queue Manager” on page 107
“Unknown Transmission Queue” on page 109

Start and stop events

Start and stop events (start only for OS/390) indicate that a queue manager has been started or has been requested to stop or quiesce.

Stop events are not recorded unless the default message-persistence of the SYSTEM.ADMIN.QMGR.EVENT queue is defined as persistent.

For more information about the event data returned in the start and stop event messages, see:
“Queue Manager Active” on page 87
“Queue Manager Not Active” on page 88

Channel events

These events are reported by channels as a result of conditions detected during their operation. For example, when a channel instance is stopped. Channel events are generated:

- By a command to start or stop 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.

Notes:

1. No channel events are generated when using MQSeries for OS/390 with distributed queuing provided by CICS.
2. Client connections on MQSeries for OS/390 Version 2, MQSeries Version 5, or later products, 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, runmqchl, or by a queue manager trigger message does not generate an event; in this case the only event generated is when the channel instance starts.

A successful start or stop channel command generates at least two events. The events are generated for both queue managers that are connected by the channel, unless one of the queue managers does not support events, for example versions of MQSeries for AS/400 previous to V3R2. Channel event messages are put onto the SYSTEM.ADMIN.CHANNEL.EVENT queue, if it is available. Otherwise, they are ignored.

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

For more information about the event data returned in the channel event messages, see:
“Channel Activated” on page 51
“Channel Auto-definition Error” on page 52

"Channel Auto-definition OK" on page 54
"Channel Conversion Error" on page 55
"Channel Not Activated" on page 58
"Channel Started" on page 60
"Channel Stopped" on page 62

IMS bridge events

These events are reported when an IMS bridge starts or stops (on OS/390 only).

For more information about the event data returned in the messages specific to IMS bridge events, see
"Bridge Started" on page 48 (OS/390 only)
"Bridge Stopped" on page 49 (OS/390 only)

Performance events

These events are notifications that a threshold condition has been reached by a resource. For example, a queue depth limit has been reached. For further details on performance events, see "Chapter 2. Understanding performance events" on page 15

Performance events are related to conditions that can affect the performance of applications that use a specified queue.

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

Performance events are not generated for the event queues themselves.

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

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

MQSeries for OS/390 supports queue depth events for QSGDISP (SHARED) queues. Service interval events are not supported. Queue manager and channel events remain unaffected by shared queues.

There are two types of performance event:

Queue depth events

Queue depth events are related to the number of messages on a queue; that is how full, or empty, the queue is. This type of event is supported for shared queues.

Queue service interval events

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

Table 1 on page 8 is a full list of events. Use it to locate information about a particular type of event message:

Types of event

Table 1. Event message data summary

Event type	*Event name*	*page*
Authority events	Not Authorized (type 1) Not Authorized (type 2) Not Authorized (type 3) Not Authorized (type 4)	72 74 76 78
Channel events	Channel Activated Channel Auto-Definition Error Channel Auto-Definition OK Channel Conversion Error Channel Not Activated Channel Started Channel Stopped Channel Stopped By User	51 52 54 55 58 60 62 65
IMS Bridge events	Bridge Started Bridge Stopped	48 49
Inhibit events	Get Inhibited Put Inhibited	71 79
Local events	Alias Base Queue Type Error Unknown Alias Base Queue Unknown Object Name	46 101 105
Performance events	Queue Depth High Queue Depth Low Queue Full Queue Service Interval High Queue Service Interval OK	81 83 85 89 91
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	67 69 93 95 97 99 103 107 109
Start and stop events	Queue Manager Active Queue Manager Not Active	87 88

Enabling and disabling events

With the exception of channel events, all instrumentation events must be enabled before they can be generated. For example, the conditions giving rise to 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.

You can enable and disable events by specifying the appropriate values for queue manager or queue attributes (or both) depending on the type of event. You do this using:

- MQSC (MQSeries) commands. For more information, see the *MQSeries MQSC Command Reference* manual.

- The operations and control panels for queue managers on OS/390. For more information, see the *MQSeries for OS/390 System Administration Guide*. Enabling and disabling an event depends on the category of the event:

Note: Attributes related to events for both queues and queue managers can be set by command only. They are not supported by the MQI call MQSET.

Enabling and disabling queue manager events

You enable queue manager events by specifying the appropriate attribute on the MQSC command ALTER QMGR. For example, to enable inhibit events on the default queue manager, use this MQSC command:

```
ALTER QMGR INHIBTEV (ENABLED)
```

To disable the event, set the INHIBTEV attribute to DISABLED using this MQSC:

```
ALTER QMGR INHIBTEV (DISABLED)
```

Authority events

You enable authority events using:
- The AUTHOREV attribute on the MQSC command ALTER QMGR

Inhibit events

You enable inhibit events using:
- The INHIBTEV attribute on the MQSC command ALTER QMGR

Local events

You enable local events using:
- The LOCALEV attribute on the MQSC command ALTER QMGR

Remote events

You enable remote events using:
- The REMOTEEV attribute on the MQSC command ALTER QMGR

Start and stop events

You enable start and stop events using:
- The STRSTPEV attribute on the MQSC command ALTER QMGR

Enabling queue manager events summary

Table 2 summarizes how to enable queue manager events:

Table 2. Enabling queue manager events using MQSC commands

Event	*Queue manager attribute*
Authority	AUTHOREV (ENABLED)
Inhibit	INHIBTEV (ENABLED)
Local	LOCALEV (ENABLED)
Remote	REMOTEEV (ENABLED)
Start and Stop	STRSTPEV (ENABLED)

Enabling channel events

Most channel events are enabled automatically and cannot be enabled or disabled by command. The exceptions are the two automatic channel definition events. However, channel events can be suppressed by not defining the channel events queue, or by making it put-inhibited. Note that this could cause a queue to fill up if remote event queues point to a put-inhibited channel events queue.

Enabling and disabling events

If a queue manager does not have a SYSTEM.ADMIN.CHANNEL.EVENT queue, or if this queue is put inhibited, all channel event messages are discarded, unless they are being put by an MCA across a link to a remote queue. In this case they are put on the dead-letter queue.

Channel auto-definition

The generation of these events is controlled by the *ChannelAutoDefEvent* queue-manager attribute. Refer to the *MQSeries Application Programming Reference* manual for further details of this attribute.

Enabling performance events

Performance events as a whole must be enabled on the queue manager, otherwise no performance events can occur. You can then enable the specific performance events by setting the appropriate queue attribute. You also have to specify the conditions that give rise to the event. For more information, see "Enabling queue service interval events" on page 18 and "Understanding queue depth events" on page 24.

Enabling queue depth events

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

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.

Enabling 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 *QServiceInterval* attribute for the queue to the appropriate length of time.

Note: When enabled, a queue service interval event can be generated only on an MQPUT call or an MQGET call. The event is **not** generated when the elapsed time becomes equal to the service interval time.

Conditions that cause events

Conditions that can give rise to instrumentation events include:

- 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.
- On the MQSeries products for AS/400 and UNIX systems, MQSeries for Digital OpenVMS, MQSeries for Tandem NonStop Kernel, and on MQSeries for Windows NT, an application attempts to open a queue specifying a user ID that is not authorized.

Note: Putting a message on the dead-letter queue can cause an event to be generated if the event conditions are met.

Event queues

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 format of messages required for transmission queues.

Shared event queues are local queues defined with the QSGDISP(SHARED) value. For more information about defining shared queues, see the *MQSeries for OS/390 System Setup Guide*.

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 will be lost. The event messages are **not**, for example, saved on the dead-letter (undelivered-message) queue.

However, the event queue may be defined as a remote queue. Then, if there is a problem on the remote system putting messages to the resolved queue, the event message **will** appear on the remote system's dead-letter queue.

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 happens whether the event message is put on the performance event queue or not. For more information about performance events changing queue attributes, see "Chapter 2. Understanding performance events" on page 15.

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 may require that an operator be informed, other events may start off an application that performs some administration tasks automatically.

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

Format of event messages

Event messages contain information about the event and its origin. Typically, these messages are processed by a system management application program tailored to meet the requirements of the enterprise at which it runs. As with all MQSeries messages, an event message has two parts: a message descriptor and the message data. The message descriptor is based on the MQMD structure, which is defined in the *MQSeries Application Programming Reference* manual. The message data is also made up of an *event header* and the *event data*. The event header contains the reason code that identifies the event type. The putting of the event message and any subsequent actions arising do not affect the reason code returned by the MQI call that caused the event. The event data provides further information about the event.

When conditions are met for an event message to be generated for a shared queue, the queue managers in the queue sharing group decide whether to generate an event message. Several queue managers can generate an event message for one shared queue, resulting in several event messages being produced. To ensure 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). For further details of the MQMD see “Message descriptor (MQMD) in event messages” on page 34.

Using event monitoring in an MQSeries network

If you write an application using events to monitor queue managers, you need to:

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 an OS/390 queue manager, you must ensure that you perform the EBCDIC to ASCII conversions.

See the *MQSeries Application Programming Guide* for more information.

Monitoring performance on Windows NT

On Windows NT, performance data is stored using performance counters that can be accessed using the system registry. Within the registry, the counters are grouped according to the type of object to which they apply. For MQSeries the type of object is `MQSeries queues`.

For each queue the following performance counters are available:
- The current queue depth
- The queue depth as a percentage of the maximum queue depth
- The number of messages per second being placed on the queue
- The number of messages per second being removed from the queue

For messages sent to a distribution list, the performance monitor counts the number of messages put on to each queue.

In the case of large messages, the performance monitor counts the appropriate number of small messages. See the *MQSeries System Administration* manual for information on using the Windows NT performance monitor to view performance information. For details of how to access the performance counters in your own application, see the Microsoft® Web site at:

`http://msdn.microsoft.com/developer/`

Follow the links from this site to obtain online platform SDK information.

Chapter 2. Understanding performance events

This chapter describes what performance events are, how they are generated, how they can be enabled, and how they are used. The chapter includes:

- "What performance events are"
- "Understanding queue service interval events" on page 16
- "Queue service interval events examples" on page 19
- "Understanding queue depth events" on page 24
- "Queue depth events examples" on page 28

In this chapter, the examples assume that you set queue attributes by using the appropriate MQSeries commands (MQSC). See the *MQSeries MQSC Command Reference* manual for more information. You can also set them using the operations and controls panels, for queue managers, on OS/390.

What performance events are

Performance events are related to conditions that can affect the performance of applications that use a specified queue.

The scope of performance events is the queue, so that MQPUT calls and MQGET calls on one queue do not affect the generation of performance events on another queue.

Note: A message must be either put on, or removed from, a queue for any performance event to be generated.

The event data contains a reason code that identifies the cause of the event, a set of performance event statistics, and other data. For more information about the event data returned in performance event messages, see:

"Queue Depth High" on page 81
"Queue Depth Low" on page 83
"Queue Full" on page 85
"Queue Service Interval High" on page 89
"Queue Service Interval OK" on page 91

Performance event statistics

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. Also, the event data contains statistics related to the event. You can use these statistics to analyze the behavior of a specified queue. Table 3 summarizes the event statistics. 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.

Table 3. Performance event statistics (continued)

Parameter	Description
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 occur:

- A performance event occurs (statistics are reset on all active queue managers).
- A queue manager stops and restarts.
- For shared queues only, the RESET QSTATS command is issued at the console (statistics are reset on the queue managers specified by the CMDSCOPE for the command).

Understanding queue service interval events

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

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

What queue service interval events are

There are two types of queue service interval event:

- A **Queue Service Interval OK** event, which indicates that, following an MQPUT call or an MQGET call that leaves a non-empty queue, an MQPUT call or an MQGET call was performed within a user-defined time period, known as the *service interval*.

  In this section, Queue Service Interval OK events are referred to as OK events.
- A **Queue Service Interval High** event, which indicates that, following an MQGET or MQPUT call that leaves a non-empty queue, an MQGET call was not performed within the user-defined service interval.

  This event message can be caused by an MQPUT call or an MQGET call.

  In this section, Queue Service Interval High events are referred to as high events.

To enable both Queue Service Interval OK and Queue Service Interval High events you need to 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.

These events are mutually exclusive, which means that if one is enabled the other is disabled. However, both events can be simultaneously disabled.

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

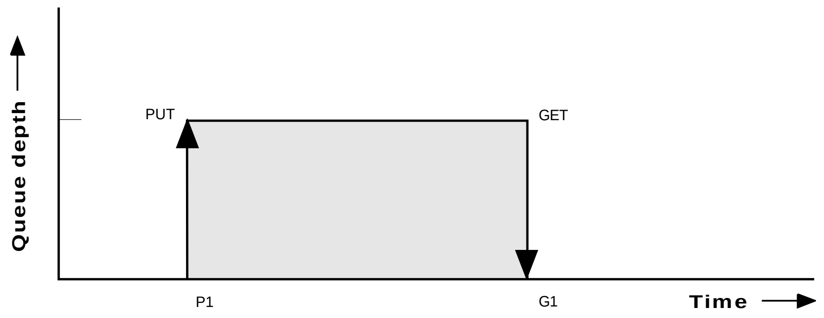


Figure 3. Understanding queue service interval events

In terms of queue service interval events, these are the possible outcomes:

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

The actual algorithm for starting the service timer and generating events is described in "Queue service interval events algorithm" on page 18.

Understanding 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 one or other of the queue service interval events 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 an MQGET call 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 an MQGET call. 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 an MQGET call 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 the operation is an MQGET call 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. The reset can be caused by a queue depth event.

Queue service interval events algorithm

This section gives the formal rules associated with the timer and the queue service interval events.

Service timer

The service timer is reset to zero and restarted:

- Following an MQPUT call to an empty queue.
- Following 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 an MQGET call, the timer is put into an OFF state.

Queue Service Interval High events

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

If the service time is greater than the service interval, an event is generated on the next MQPUT or MQGET call.

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 the next MQGET call.

Enabling queue service interval events

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

1. Enable performance events on the queue manager, using the queue manager attribute *`PerformanceEvent`* (PERFMEV in MQSC).
2. Set the control attribute, *`QServiceIntervalEvent`*, for a Queue Service Interval High or OK event on the queue, as required (QSVCIEV in MQSC).
3. Specify the service interval time by setting the *`QServiceInterval`* attribute for the queue to the appropriate length of time (QSVCINT in MQSC).

For 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)
```

Note: When enabled, a queue service interval event can only be generated on an MQPUT call or an MQGET call. The event is **not** generated when the elapsed time becomes equal to the service interval time.

Automatic enabling of queue service interval events

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.

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

Notes:

All performance events must be enabled using the queue manager attribute PERFMEV.

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.

Queue service interval events examples

This section provides progressively more complex examples to illustrate the use of queue service interval events.

The figures accompanying the examples have the same structure:

- The top section is a graph of queue depth against time, showing individual MQGET calls and MQPUT calls.
- The middle 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 bottom section of each figure shows which events are enabled at any instant and what events are generated.

The following examples illustrate:

- 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.

Example 1 (queue service interval events)

This example shows a simple sequence of MQGET calls and MQPUT calls, where the queue depth is always one or zero.

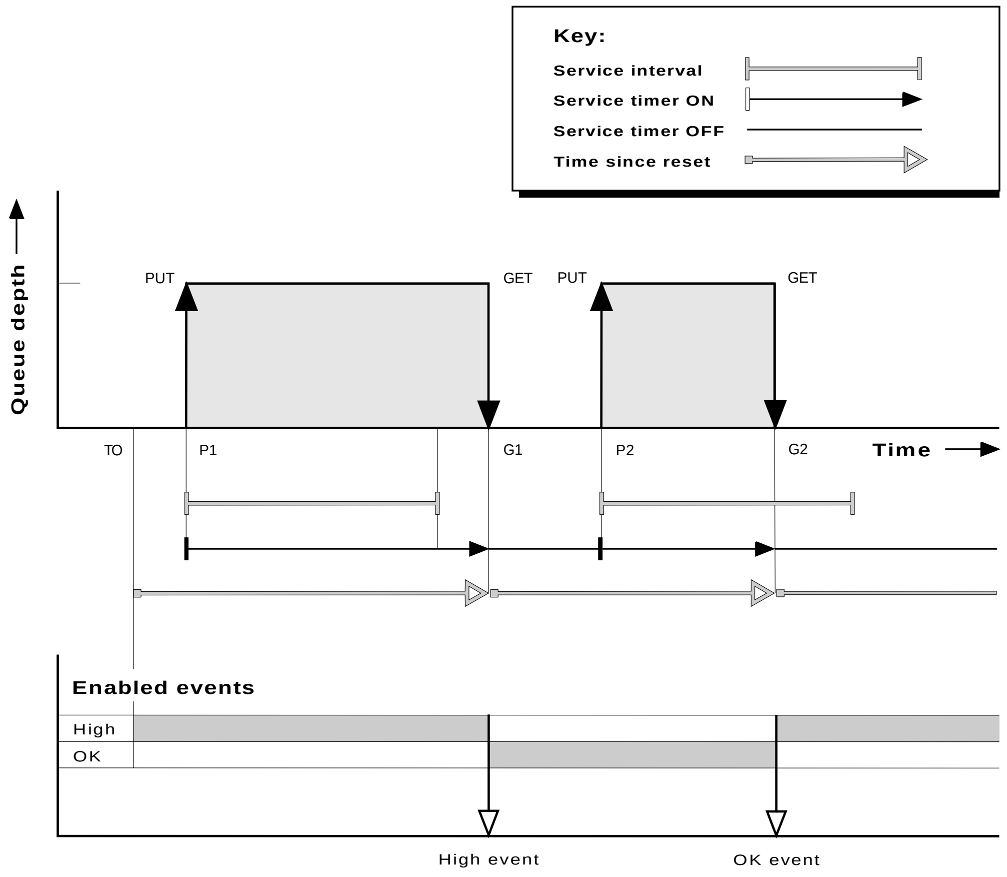


Figure 4. Queue service interval events - example 1

Commentary

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

   Note that T0 may 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 for example 1

Table 5 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_{P2}
HighQDepth	1	1
MsgEnqCount	1	1
MsgDeqCount	1	1

The middle part of Figure 4 on page 20 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 will 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 will occur 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 will occur on the next get.

What queue service interval events tell you

You must exercise some caution when you look at queue statistics. Figure 4 on page 20 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.

Example 2 (queue service interval events)

This example illustrates a sequence of MQPUT calls and MQGET calls, where the queue depth is not always one or zero. It also shows instances of the timer being reset without events being generated, for example, at T_{P2}.

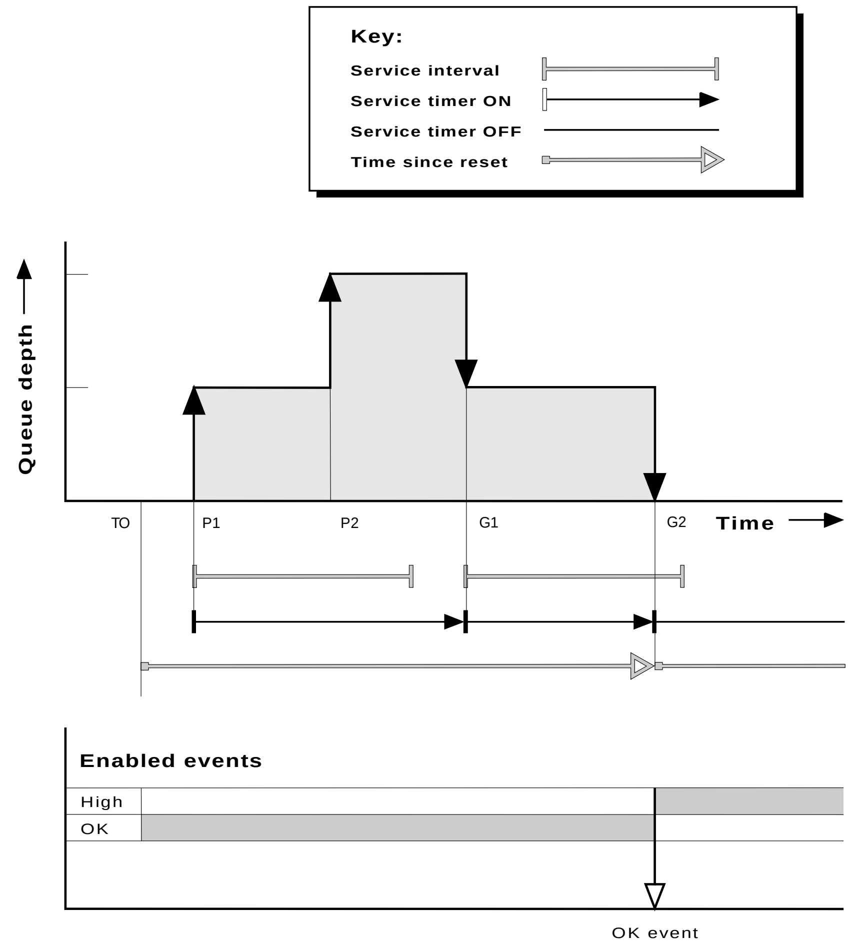


Figure 5. Queue service interval events - example 2

Commentary

In this example, **OK events are enabled** initially and queue statistics were reset at T_0.

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 for example 2

Table 6 summarizes the event statistics for this example.

Table 6. Event statistics summary for example 2

Time of event	T_{G2}
Type of event	OK

Table 6. Event statistics summary for example 2 (continued)

TimeSinceReset	T_{G2} - T_0
HighQDepth	2
MsgEnqCount	2
MsgDeqCount	2

Example 3 (queue service interval events)

This example shows a sequence of MQGET calls and MQPUT calls that is more sporadic than the previous examples.

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.

Queue service interval events

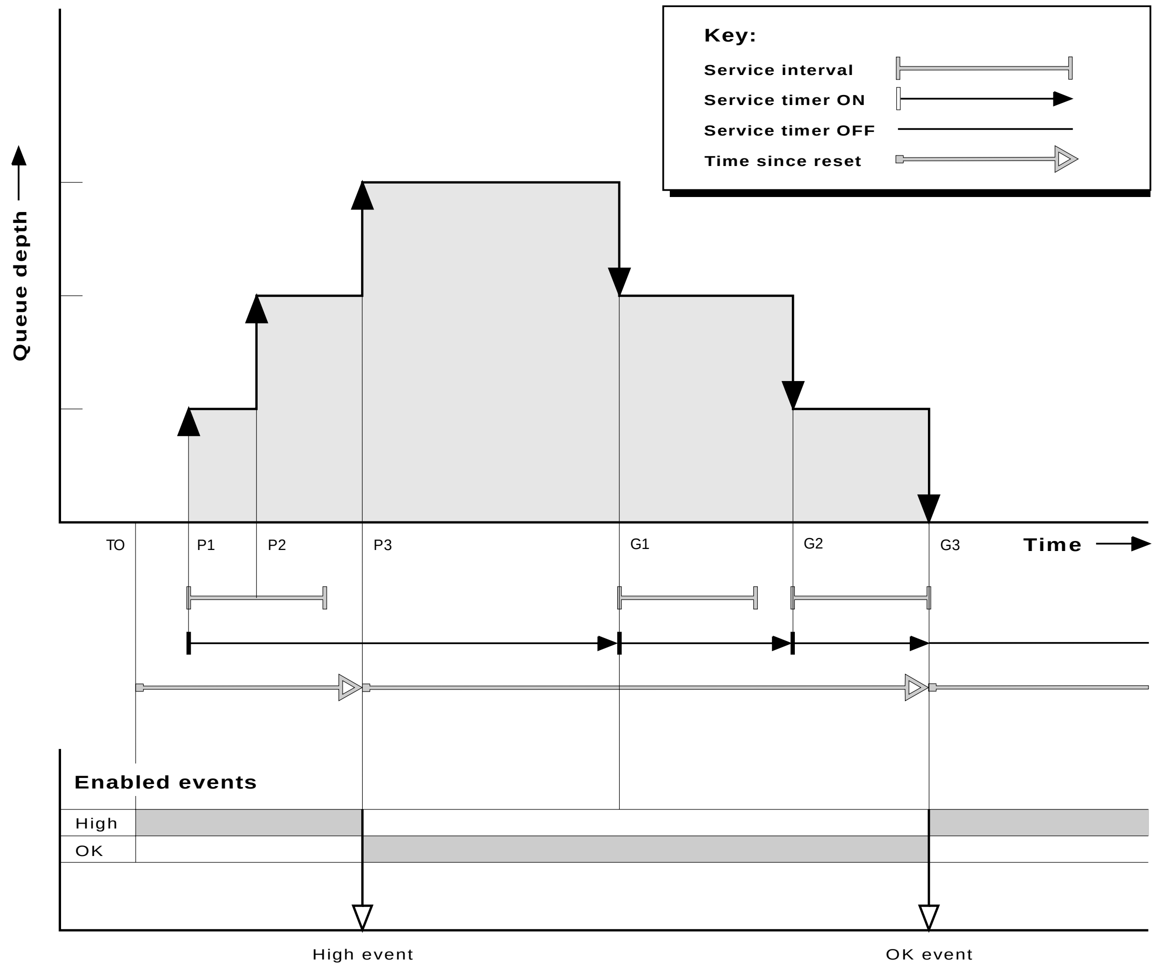


Figure 6. Queue service interval events - example 3

Event statistics summary for example 3

The following table summarizes the statistics returned in the event message data, for each event in 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

Understanding queue depth events

In MQSeries 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, it can be a 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 may 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.

Clearly, having advanced warning that problems may be on their way makes it easier to take preventive action. For this purpose, queue depth events are provided.

What queue depth events are

Queue depth events are related to the queue depth, that is, the number of messages on the queue. The types of queue depth events are:
- **Queue Depth High events**, which indicate that the queue depth has increased to a predefined threshold called the Queue Depth High limit.
- **Queue Depth Low events**, which indicate that the queue depth has decreased to a predefined threshold called the Queue Depth Low limit.
- **Queue Full events**, which 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 should take some preventive action. If this action is successful and the queue depth drops to a ‘safe’ level, the queue manager can be configured to generate a Queue Depth Low event indicating an ‘all clear’ state.

Figure 8 on page 29 shows a graph of queue depth against time in such a case. The preventive action was (presumably) taken between T_2 and T_3 and continues to have effect until T_4 when the queue depth is well inside the ‘safe’ zone.

Shared queues and queue depth events (MQSeries for OS/390)

When a queue depth event occurs on a shared queue, the queue managers in the queue-sharing group produce an event message, if the queue manager attribute *PerformanceEvent* (PERFMEV in MQSC) is set to ENABLED. If PERFMEV is set to DISABLED on some of the queue managers, event messages are not produced by those queue managers, making event monitoring from an application more difficult. To avoid this, give each queue manager the same setting for the *PerformanceEvent* attribute. This event message represents the individual usage of the shared queue by each queue manager. If a queue manager performs no activity on the shared queue, various values in the event message are null or zero. Null event messages:
- Allow you to ensure there is one event message for each active queue manager in a queue-sharing group
- Can highlight cases where there has been no activity on a shared queue for a queue manager that produced the event message

Enabling queue depth events

By default, all queue depth events are disabled. To configure a queue for any of the queue depth events you must:
1. Enable performance events on the queue manager, using the queue manager attribute *PerformanceEvent* (PERFMEV in MQSC).

2. Enable the event on the required queue by setting the following as required:
   - *QDepthHighEvent*(QDPHIEV in MQSC)
   - *QDepthLowEvent*(QDPLOEV in MQSC)
   - *QDepthMaxEvent*(QDPMAXEV in MQSC)
3. Set the limits, if required, to the appropriate levels, expressed as a percentage of the maximum queue depth, by setting either:
   - *QDepthHighLimit*(QDEPTHHI in MQSC), and
   - *QDepthLowLimit*(QDEPTHLO in MQSC).

Enabling queue depth events on shared queues (MQSeries for OS/390)

When a queue manager determines that an event should be issued, the shared queue object definition is updated to toggle the active performance event attributes. For example, depending on the definition of the queue attributes, a Queue Depth High event enables a Queue Depth Low and a Queue Full event. After the shared queue object has been updated successfully, the queue manager that detected the performance event initially becomes the *coordinating queue manager*.

The **coordinating queue manager**:

1. Determines if it has performance events enabled.
2. If it does, issues an event message that captures all shared queue performance data it has gathered since the last time an event message was created, or since the queue statistics were last reset. The message descriptor (MQMD) of this message contains a unique correlation identifier (*CorrelId*) created by the coordinating queue manager.
3. Broadcasts to all other **active** queue managers in the same queue-sharing group to request the production of an event message for the shared queue. The broadcast contains the correlation identifier created by the coordinating queue manager for the set of event messages.

After receiving a request from the coordinating queue manager, an **active queue manager in a queue-sharing group**:

1. Determines if its PERFMEV is ENABLED.
2. If it is, the active queue manager issues an event message for the shared queue, recording all operations performed by the receiving (active) queue manager since the last time an event message was created, or since the last statistics reset. The message descriptor (MQMD) of this event message contains the unique correlation identifier (*CorrelId*) specified by the coordinating queue manager.

When performance events occur on a shared queue, *n* event messages are produced, where *n* is 1 to the number of active queue managers in the queue-sharing group. Each event message contains data that relates to the shared queue activity for the queue manager where the event message was generated.

You can view event message data for a shared queue using the:

- Queue-sharing view.

  All data from event messages with the same correlation identifier is collected here.
- Queue manager view.

  Each event message shows how much it has been used by its originating queue manager.

Differences between shared and nonshared queues: Enabling queue depth events on shared queues differs from enabling events on nonsharedqueues. A key difference is that events are switched for shared queues even if PERFMEV is DISABLED on the queue manager. This is not the case for nonshared queues.

Consider the following example which illustrates this difference.

- QM1 is a queue manager with `PerformanceEvent` (PERFMEV in MQSC) set to DISABLED.
- SQ1 is a shared queue with QSGDISP set to (SHARED) QLOCAL in MQSC.
- LQ1 is a nonshared queue with QSGDISP set to (QMGR) QLOCAL in MQSC.

Both queues have the following attributes set on their definitions:

- QDPHIEV (ENABLED)
- QDPLOEV (DISABLED)
- QDPMAXEV (DISABLED)

If messages are placed on both queues so that the depth meets or exceeds the QDEPTHHI threshold, the QDPHIEV value on SQ1 switches to DISABLED. Also, QDPLOEV and QDPMAXEV are switched to ENABLED. SQ1's attributes are automatically switched for each performance event at the time the event criteria are met.

In contrast the attributes for LQ1 remain unchanged until PERFMEV on the queue manager is ENABLED. This means that if the queue manager's PERFMEV attribute is ENABLED, DISABLED and then re-ENABLED for instance, the performance event settings on shared queues might not be consistent with those of nonshared queues, even though they might have initially been the same.

Enabling Queue Depth High events

When enabled, 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.

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)
```

Automatically enabling Queue Depth High events: 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.

Enabling Queue Depth Low events

When enabled, a Queue Depth Low event is generated when a message is removed from a queue by an MQGET call operation causing the queue depth to be less than or equal to the value determined by the Queue Depth Low limit.

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)
```

Queue depth events

Automatically enabling Queue Depth Low events: 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.

Enabling Queue Full events

When enabled, a Queue Full event is generated when an application is unable to put a message onto a queue because the queue is full.

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

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

Automatically enabling Queue Full events: 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.

Table 8. Enabling queue depth events using MQSC

Queue depth event	*Queue attributes*
Queue depth high	QDPHIEV (ENABLED) QDEPTHHI (*hh*) where *hh* is the queue depth high limit.
Queue depth low	QDPLOEV (ENABLED) QDEPTHLO (*ll*) where *ll* is the Queue depth low limit. (Both values are expressed as a percentage of the maximum queue depth, which is specified by the queue attribute `MAXDEPTH`.)
Queue full	QDPMAXEV (ENABLED)

Notes: All performance events must be enabled using the queue manager attribute PERFMEV.

Queue depth events examples

This section contains some examples of queue depth events. The following examples illustrate how queue depth varies over time.

Example 1 (queue depth events)

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

The MQSeries 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)
```

Figure 7. Definition of MYQUEUE1

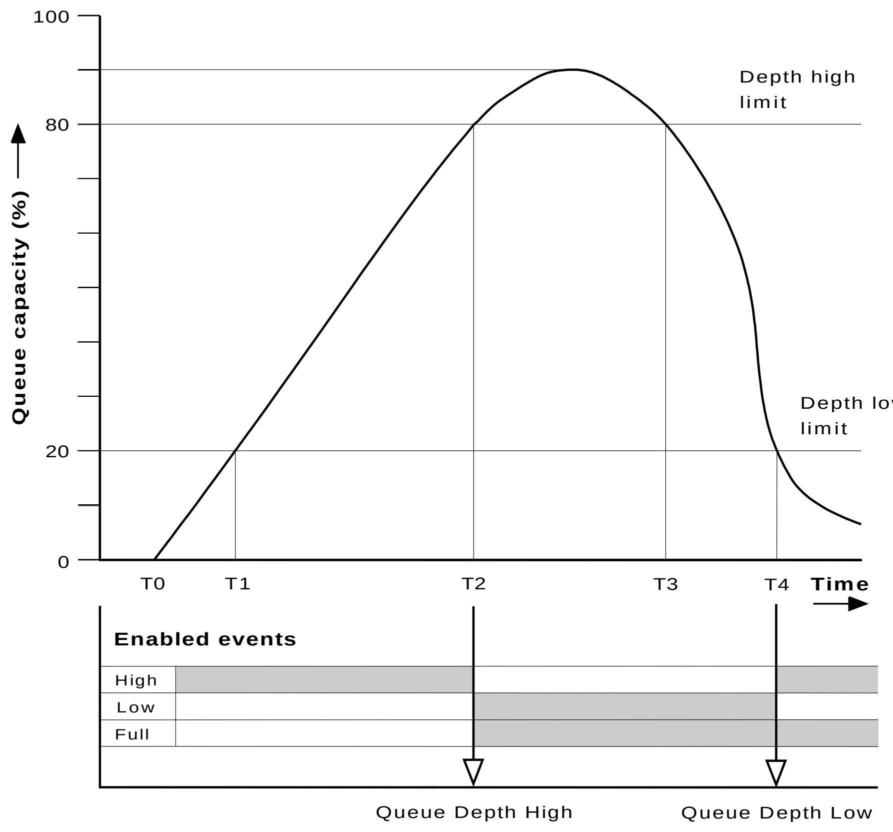


Figure 8. Queue depth events (1)

Commentary

Figure 8 shows how the queue depth changes over time:

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.

Queue depth events

Table 9 summarizes the queue event statistics and Table 10 summarizes which events are enabled at different times for this example.

Table 9. 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 10. 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

Example 2 (queue depth events)

This is a more extensive example. However, the principles remain the same. This example assumes the use of the same queue MYQUEUE1 as defined in Figure 7 on page 29.

Table 11 on page 32 summarizes the queue event statistics and Table 12 on page 32 summarizes which events are enabled at different times for this example.

Figure 9 on page 31 shows the variation of queue depth over time.

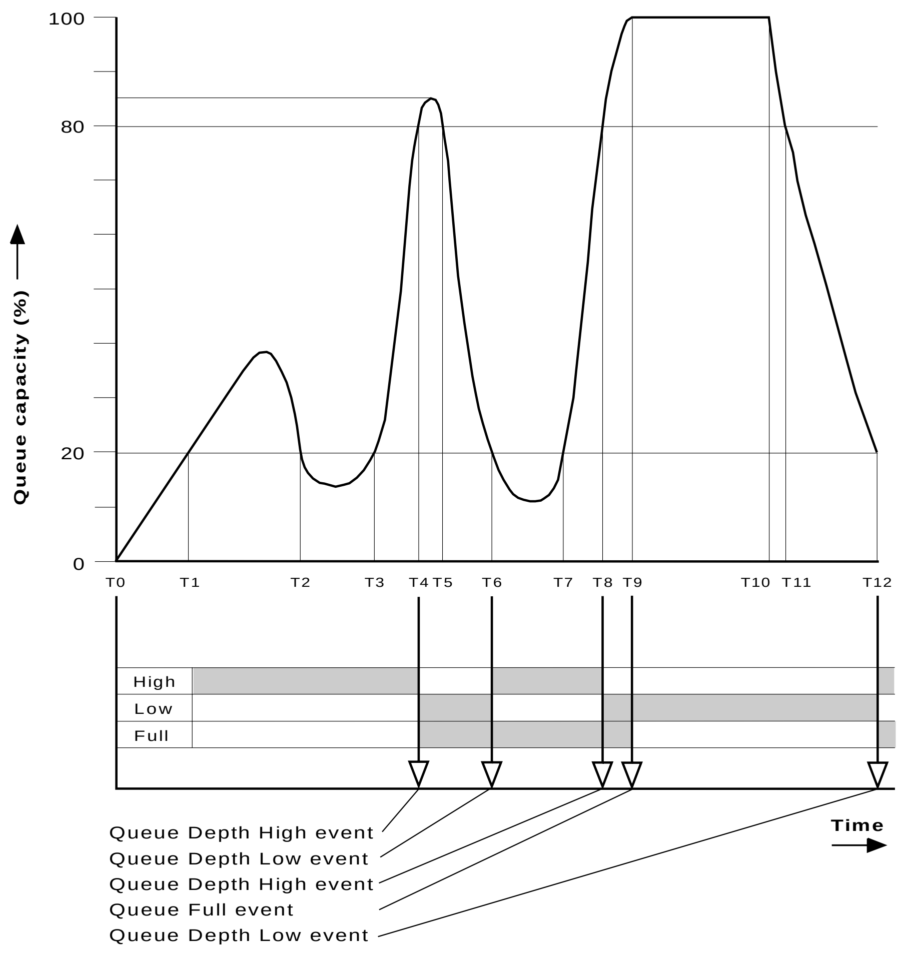


Figure 9. Queue depth events (2)

Commentary

Some points to note are:

1. No Queue Depth Low event is generated at:
   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 (example 2)

Table 11. 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 12. 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.

Chapter 3. Event message reference

This chapter provides an overview of the event message format. It describes the information returned in the event message for each instrumentation event, including returned parameters.

The chapter includes:

Event message format

Event messages are standard MQSeries messages containing a message descriptor and message data.

Table 13 on page 34 shows the basic structure of these messages, and the names of the fields in an event message for queue service interval events.

Event message format

Table 13. Event message structure for queue service interval events

Message descriptor	Message data	
MQMD structure [1]	Event header MQCFH structure [2]	Event data [3]
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 number Command identifier (event type) Message sequence number Control options Completion code Reason code (MQRC_*) Parameter count	Queue manager name Queue name Time since last reset Maximum number of messages on the queue Number of messages put on the queue Number of messages taken off the queue

Notes:
1. MQMD is the standard structure for MQSeries message headers.
2. MQCFH is the standard structure for an event header.
3. The parameters shown are those returned for a queue service interval event. The actual event data depends on the specific event.

In general, you need only a subset of this information for any system management programs that you write. For example, your application might need the following data:
- The name of the application causing the event
- The name of the queue manager on which the event occurred
- The queue on which the event was generated
- The event statistics

Message descriptor (MQMD) in event messages

The format of the message descriptor is defined by the MQSeries MQMD data structure, which is found in all MQSeries messages and is described in the *MQSeries Application Programming Reference* manual. The message descriptor contains information that can be used by a user-written system monitoring application. For example:
- The message type
- The format type

- The date and time that the message was put on the event queue

In particular, the information in the descriptor informs a system management application that the message type is MQMT_DATAGRAM, and the message format is MQFMT_EVENT.

In an event message, many of these fields contain fixed data, which is supplied by the queue manager that generated the message. The fields that make up the MQMD structure are described in “MQMD (Message descriptor)”. 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 event message was put on the event queue.

Message data in event messages

The event message data is based on the programmable command format (PCF), which is used in PCF command inquiries and responses.

The event message consists of two parts: the event header and the event data.

Event header (MQCFH)

The information in MQCFH specifies:

- If the message is an event message.
- The category of event, that is, whether the event is a queue manager, performance, or channel event.
- A reason code specifying the cause of the event. For events caused by MQI calls, this reason code is the same as the reason code for the MQI call.

Reason codes have names that begin with the characters MQRC_. For example, the reason code MQRC_PUT_INHIBITED is generated when an application attempts to put a message on a queue that is not enabled for puts. MQCFH is described in “MQCFH (Event header)” on page 41.

Event data

See “Event message descriptions” on page 45.

MQMD (Message descriptor)

The MQMD structure describes the information that accompanies the message data of an event message. For a full description of MQMD, including a description of the elementary datatype of each parameter, see the *MQSeries Application Programming Reference* manual.

For an event message, the MQMD structure contains these values:

StrucId

Description:	Structure identifier.
Datatype:	MQCHAR4.
Initial value:	MQMD_STRUC_ID

Valid value:

MQMD_STRUC_ID
Identifier for message descriptor structure.

For the C programming language, the constant MQMD_STRUC_ID_ARRAY is also defined; this has the same value as MQMD_STRUC_ID, but is an array of characters not a string.

Version

Description:	Structure version number.
Datatype:	MQLONG.
Initial value:	MQMD_VERSION_1.

Valid values:

MQMD_VERSION_1
Version-1 message descriptor structure, supported in all environments.

MQMD_VERSION_2
Version-2 message descriptor structure, supported on AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows NT client, Windows NT, and Windows 2000 platforms.

Report

Description:	Options for report messages.
Datatype:	MQLONG.
Initial value:	MQRO_NONE.

Valid values:

MQRO_NONE
No reports required.

MsgType

Description:	Indicates type of message.
Datatype:	MQLONG.
Initial value:	MQMT_DATAGRAM.

Valid values:

MQMT_DATAGRAM
Message does not require a reply.

Expiry

Description:	Message lifetime.
Datatype:	MQLONG.
Initial value:	MQEI_UNLIMITED.

Valid values:

MQEI_UNLIMITED
The message does not have an expiry time.

Feedback

Description:	Feedback or reason code.
Datatype:	MQLONG.

Initial value:	MQFB_NONE.
Valid values:	MQFB_NONE No feedback provided.

Encoding

Description:	Numeric encoding of message data.
Datatype:	MQLONG.
Initial value:	MQENC_NATIVE.
Valid values:	MQENC_NATIVE The default for the programming language and machine on which the application is running.

CodedCharSetId

Description:	Character set identifier of event message data.
Datatype:	MQLONG.
Initial value:	MQCCSI_Q_MGR.
Valid values:	MQCCSI_Q_MQR Coded character set ID (CCSID) of the queue manager generating the event.

Format

Description:	Format name of message data.
Datatype:	MQCHAR8.
Initial value:	MQFMT_NONE.
Valid value:	MQFMT_EVENT Event message. For the C programming language, the constant MQFMT_EVENT_ARRAY is also defined; this has the same value as MQFMT_EVENT, but is an array of characters not a string.

Priority

Description:	Message priority.
Datatype:	MQLONG.
Initial value:	MQPRI_PRIORITY_AS_Q_DEF.
Valid values:	MQPRI_PRIORITY_AS_Q_DEF Default priority of the event queue, if it is a local queue, or its local definition at the queue manager generating the event.

Persistence

Description:	Message persistence.
Datatype:	MQLONG.
Initial value:	MQPER_PERSISTENCE_AS_Q_DEF.

Valid values:	MQPER_PERSISTENCE_AS_Q_DEF Default persistence of the event queue, if it is a local queue, or its local definition at the queue manager generating the event.

MsgId

Description:	Message identifier.
Datatype:	MQBYTE24.
Initial value:	MQMI_NONE.
Valid values:	A unique value generated by the queue manager.

CorrelId

Description:	Correlation identifier. For events on a shared queue, this parameter is set, so you can track multiple event messages from different queue managers
Datatype:	MQBYTE24.
Initial value:	MQCI_NONE.
Valid values:	MQCI_NONE No correlation identifier is specified. This is for local queues only. For the C programming language, the constant MQCI_NONE_ARRAY is also defined; this has the same value as MQCI_NONE, but is an array of characters not a string. For shared queues: a nonzero correlation identifier is specified. The characters are specified below: 1–4 Product identifier (’CSQ ’) 5–8 Queue-sharing group name 9 Queue manager identifier 10–17 Time stamp 18–24 Nulls

BackoutCount

Description:	Backout counter.
Datatype:	MQLONG.
Initial value:	0.
Valid values:	0.

ReplyToQ

Description:	Name of reply queue.
Datatype:	MQCHAR48.
Initial value:	Null string in the C programming language, or 48 blank characters in other programming languages.
Valid values:	Always blank.

ReplyToQMgr

Description:	Name of reply queue manager.
Datatype:	MQCHAR48.

Initial value:	Null string in the C programming language, or 48 blank characters in other programming languages.
Valid values:	The queue manager name at the originating system.

UserIdentifier

Description:	Identifies the application that originated the message.
Datatype:	MQCHAR12.
Initial value:	Null string in the C programming language, or 12 blank characters in other programming languages.
Valid values:	Always blank.

AccountingToken

Description:	Accounting token that allows an application to charge for work done as a result of the message.
Datatype:	MQBYTE32.
Initial value:	MQACT_NONE.
Valid values:	MQACT_NONE No accounting token is specified. For the C programming language, the constant MQACT_NONE_ARRAY is also defined; this has the same value as MQACT_NONE, but is an array of characters not a string.

ApplIdentityData

Description:	Application data relating to identity.
Datatype:	MQCHAR32.
Initial value:	Null string in the C programming language and 32 blank characters in other programming languages.
Valid values:	Always blank.

PutApplType

Description:	Type of application that put the message.
Datatype:	MQLONG.
Initial value:	MQAT_NO_CONTEXT.
Valid values:	MQAT_QMGR Queue-manager-generated message.

PutApplName

Description:	Name of application that put the message.
Datatype:	MQCHAR28.
Initial value:	Null string in the C programming language and 28 blank characters in other programming languages.
Valid values:	Dependent on the environment.

PutDate

Description:	Date when message was put.

Datatype:	MQCHAR8.
Initial value:	Null string in the C programming language and 8 blank characters in other programming languages.
Valid values:	As generated by the queue manager.

PutTime

Description:	Time when message was put.
Datatype:	MQCHAR8.
Initial value:	Null string in the C programming language and 8 blank characters in other programming languages.
Valid values:	As generated by the queue manager.

ApplOriginData

Description:	Application data relating to origin.
Datatype:	MQCHAR4.
Initial value:	Null string in the C programming language and 4 blank characters in other programming languages.
Valid values:	Always blank.

Note: 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.
Datatype:	MQBYTE24.
Initial value:	MQGI_NONE.
Valid values:	MQGI_NONE No group identifier specified. For the C programming language, the constant MQGI_NONE_ARRAY is also defined; this has the same value as MQGI_NONE, but is an array of characters not a string.

MsgSeqNumber

Description:	Sequence number of logical message within group.
Datatype:	MQLONG.
Initial value:	1.
Valid values:	1.

Offset

Description:	Offset of data in physical message from start of logical message.
Datatype:	MQLONG.
Initial value:	0.
Valid values:	0.

MsgFlags

Description:	Message flags that specify attributes of the message or control its processing.

Datatype:	MQLONG.
Initial value:	MQMF_NONE.
Valid values:	MQMF_NONE No message flags (default message attributes).

OriginalLength

Description:	Length of original message.
Datatype:	MQLONG.
Initial value:	MQOL_UNDEFINED.
Valid values:	MQOL_UNDEFINED Original length of message not defined.

MQCFH (Event header)

The MQCFH structure is the header structure used for event messages and for PCF messages. When the structure is used for event messages, the message descriptor *Format* field is MQFMT_EVENT. The datatype of the following parameters (MQLONG) is described in the *MQSeries Application Programming Reference* manual.

For an event, the MQCFH structure contains these values:

Type

Description:	Structure type that identifies the content of the message.
Datatype:	MQLONG.
Initial value:	MQCFT_COMMAND.
Valid values:	MQCFT_EVENT Message is reporting an event.

StrucLength

Description:	Structure length.
Datatype:	MQLONG.
Initial value:	MQCFH_STRUC_LENGTH.
Valid values:	MQCFH_STRUC_LENGTH Length in bytes of MQCFH structure.

Version

Description:	Structure version number.
Datatype:	MQLONG.
Initial value:	MQCFH_VERSION_1.
Valid values:	MQCFH_VERSION_1 Version number for command format header structure.

Event header

Command

Description:	Command identifier. This identifies the event category.
Datatype:	MQLONG.
Initial value:	0.
Valid values:	MQCMD_Q_MGR_EVENT Queue manager event. MQCMD_PERFM_EVENT Performance event. MQCMD_CHANNEL_EVENT Channel event.

MsgSeqNumber

Description:	Message sequence number. This is the sequence number of the message within a group of related messages.
Datatype:	MQLONG.
Initial value:	1.
Valid values:	1.

Control

Description:	Control options.
Datatype:	MQLONG.
Initial value:	MQCFC_LAST.
Valid values:	MQCFC_LAST Last or only message in the group.

CompCode

Description:	Completion code.
Datatype:	MQLONG.
Initial value:	MQCC_OK.
Valid values:	MQCC_OK Event reporting OK condition. MQCC_WARNING Event reporting warning condition. All events have this completion code, unless otherwise specified.

Reason

Description:	Reason code qualifying completion code.
Datatype:	MQLONG.
Initial value:	MQRC_NONE.
Valid values:	MQRC_* Dependent on the event being reported. **Note:** Events with the same reason code are further identified by the *ReasonQualifier* parameter in the event data.

ParameterCount

Description:	Count of parameter structures. This is the number of parameter structures (MQCFIN and MQCFST) that follow the MQCFH structure.
Datatype:	MQLONG.
Initial value:	0.
Valid values:	0 or greater.

C language declaration

```
typedef struct tagMQCFH {
  MQLONG  Type;            /* Structure type */
  MQLONG  StrucLength;     /* Structure length */
  MQLONG  Version;         /* Structure version number */
  MQLONG  Command;         /* Command identifier */
  MQLONG  MsgSeqNumber;    /* Message sequence number */
  MQLONG  Control;         /* Control options */
  MQLONG  CompCode;        /* Completion code */
  MQLONG  Reason;          /* Reason code qualifying completion code */
  MQLONG  ParameterCount;  /* Count of parameter structures */
 } MQCFH;
```

COBOL language declaration

```
**   MQCFH structure
  10 MQCFH.
**    Structure type
   15 MQCFH-TYPE           PIC S9(9) BINARY.
**    Structure length
   15 MQCFH-STRUCLENGTH    PIC S9(9) BINARY.
**    Structure version number
   15 MQCFH-VERSION        PIC S9(9) BINARY.
**    Command identifier
   15 MQCFH-COMMAND        PIC S9(9) BINARY.
**    Message sequence number
   15 MQCFH-MSGSEQNUMBER   PIC S9(9) BINARY.
**    Control options
   15 MQCFH-CONTROL        PIC S9(9) BINARY.
**    Completion code
   15 MQCFH-COMPCODE       PIC S9(9) BINARY.
**    Reason code qualifying completion code
   15 MQCFH-REASON         PIC S9(9) BINARY.
**    Count of parameter structures
   15 MQCFH-PARAMETERCOUNT PIC S9(9) BINARY.
```

PL/I language declaration (AIX, OS/2, OS/390, and Windows NT)

```
dcl
 1 MQCFH based,
  3 Type           fixed bin(31), /* Structure type */
  3 StrucLength    fixed bin(31), /* Structure length */
  3 Version        fixed bin(31), /* Structure version number */
  3 Command        fixed bin(31), /* Command identifier */
  3 MsgSeqNumber   fixed bin(31), /* Message sequence number */
  3 Control        fixed bin(31), /* Control options */
  3 CompCode       fixed bin(31), /* Completion code */
  3 Reason         fixed bin(31), /* Reason code qualifying completion
                                     code */
  3 ParameterCount fixed bin(31); /* Count of parameter structures */
```

System/390® assembler-language declaration (OS/390 only)

```
MQCFH                         DSECT
MQCFH_TYPE                    DS   F        Structure type
MQCFH_STRUCLENGTH             DS   F        Structure length
MQCFH_VERSION                 DS   F        Structure version number
MQCFH_COMMAND                 DS   F        Command identifier
MQCFH_MSGSEQNUMBER            DS   F        Message sequence number
MQCFH_CONTROL                 DS   F        Control options
MQCFH_COMPCODE                DS   F        Completion code
MQCFH_REASON                  DS   F        Reason code qualifying
*                                           completion code
MQCFH_PARAMETERCOUNT          DS   F        Count of parameter
*                                           structures
MQCFH_LENGTH                  EQU  *-MQCFH  Length of structure
                              ORG  MQCFH
MQCFH_AREA                    DS   CL(MQCFH_LENGTH)
```

Visual Basic® language declaration (Windows platforms only)

```
Type MQCFH
  Type As Long             'Structure type
  StrucLength As Long      'Structure length
  Version As Long          'Structure version number
  Command As Long          'Command identifier
  MsgSeqNumber As Long     'Message sequence number
  Control As Long          'Control options
  CompCode As Long         'Completion code
  Reason As Long           'Reason code qualifying completion code
  ParameterCount As Long  'Count of parameter structures
End Type

Global MQCFH_DEFAULT As MQCFH
```

Event message descriptions

The event message data contains information specific to the event. This includes the name of the queue manager and, where appropriate, the name of the queue.

The data structures returned depend on which particular event was generated. In addition, for some events, certain of the structures are optional, and are returned only if they contain information that is relevant to the circumstances giving rise to the event. The values in the data structures depend on the circumstances that caused the event to be generated.

Notes

1. The event structures in the event data are not returned in a defined order. They must be identified from the parameter identifiers shown in the description.
2. The events described in the reference section are available on all platforms, unless specific limitations are shown at the start of an event.
3. The structure datatypes of each parameter are described in “Appendix A. Structure datatypes MQCFIN and MQCFST” on page 123.
4. Version 2.0 of MQSeries for Windows does not generate MQSeries events.

Alias Base Queue Type Error

Event name:	Alias Base Queue Type Error.
Reason code in MQCFH:	MQRC_ALIAS_BASE_Q_TYPE_ERROR (2001, X'7D1'). Alias base queue not a valid type.
Event description:	An MQOPEN or MQPUT1 call was issued specifying an alias queue as the destination, but the *BaseQName* in the alias queue definition resolves to a queue that is not a local queue, or local definition of a remote queue.
Event type:	Local.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

BaseQName

Description:	Queue name to which the alias resolves.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

QType

Description:	Type of queue to which the alias resolves.
Identifier:	MQIA_Q_TYPE.
Datatype:	MQCFIN.
Values:	MQQT_ALIAS Alias queue definition. MQQT_MODEL Model queue definition.
Returned:	Always.

ApplType

Description:	Type of the application making the call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, rather than the client.

Bridge Started

Event name:	Bridge Started.
Reason code in MQCFH:	MQRC_BRIDGE_STARTED (2125, X'84D'). Bridge started.
Event description:	The IMS bridge has been started.
Event type:	IMS Bridge.
Platforms:	MQSeries for OS/390 only.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Data type:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

BridgeType

Description:	Bridge type.
Identifier:	MQIACF_BRIDGE_TYPE.
Data type:	MQCFIN.
Values:	MQBT_OTMA OTMA bridge.
Returned:	Always.

BridgeName

Description:	Bridge name. For bridges of type MQBT_OTMA, the name is of the form `XCFgroupXCFmember`, where `XCFgroup` is the XCF group name to which both IMS and MQSeries belong. `XCFmember` is the XCF member name of the IMS system.
Identifier:	MQCACF_BRIDGE_NAME.
Data type:	MQCFST.
Maximum length:	MQ_BRIDGE_NAME_LENGTH.
Returned:	Always.

Bridge Stopped

Event name:	Bridge Stopped.
Reason code in MQCFH:	MQRC_BRIDGE_STOPPED (2126, X'84E'). Bridge stopped.
Event description:	The IMS bridge has been stopped.
Event type:	IMS Bridge.
Platforms:	MQSeries for OS/390 only.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier that qualifies the reason code in MQCFH.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_BRIDGE_STOPPED_OK Bridge has been stopped with either a zero return code or a warning return code. For MQBT_OTMA bridges, one side or the other issued a normal IXCLEAVE request. MQRQ_BRIDGE_STOPPED_ERROR Bridge has been stopped but there is an error reported.
Returned:	Always.

BridgeType

Description:	Bridge type.
Identifier:	MQIACF_BRIDGE_TYPE.
Datatype:	MQCFIN.
Value:	MQBT_OTMA OTMA bridge.
Returned:	Always.

BridgeName

Description:	Bridge name. For bridges of type MQBT_OTMA, the name is of the form `XCFgroupXCFmember`, where `XCFgroup` is the XCF group name to which both IMS and MQSeries belong. `XCFmember` is the XCF member name of the IMS system.
Identifier:	MQCACF_BRIDGE_NAME.
Datatype:	MQCFST.

Maximum length:	MQ_BRIDGE_NAME_LENGTH.
Returned:	Always.

ErrorIdentifier

Description:	When a bridge is stopped due to an error, this code identifies the error. If the event reports a bridge stop failure, the IMS sense code is set.
Identifier:	MQIACF_ERROR_IDENTIFIER.
Datatype:	MQCFIN.
Returned:	If *ReasonQualifier* is MQRQ_BRIDGE_STOPPED_ERROR.

Channel Activated

Event name:	Channel Activated.
Reason code in MQCFH:	MQRC_CHANNEL_ACTIVATED (2295, X'8F7'). Channel activated.
Event description:	This condition is detected when a channel that has been waiting to become active, and for which a Channel Not Activated event has been generated, is now able to become active, because an active slot has been released by another channel. This event is not generated for a channel that is able to become active without waiting for an active slot to be released.
Event type:	Channel.
Platforms:	All, except MQSeries for OS/390 if CICS® is used for distributed queue management.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the reason code.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ChannelName

Description:	Channel Name.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCACH_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	For sender, server, cluster-sender, and cluster-receiver channels only.

ConnectionName

Description:	If the channel has successfully established a TCP connection, this is the Internet address. Otherwise it is the contents of the *ConnectionName* field in the channel definition.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Only for commands that do not contain a generic name.

Channel Auto-definition Error

Event name:	Channel Auto-definition Error.
Reason code in MQCFH:	MQRC_CHANNEL_AUTO_DEF_ERROR (2234, X'8BA'). Automatic channel definition failed.
Event description:	This condition is detected when the automatic definition of a channel fails; this may be because an error occurred during the definition process, or because the channel automatic-definition exit inhibited the definition. Additional information indicating the reason for the failure is returned in the event message.
Event type:	Channel.
Platforms:	Any MQSeries Version 5 product, except MQSeries for OS/390 when using CICS for distributed queuing.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ChannelName

Description:	Name of the channel for which the auto-definiton has failed.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

ChannelType

Description:	Channel Type. This specifies the type of channel for which the auto-definition has failed.
Identifier:	MQIACH_CHANNEL_TYPE.
Datatype:	MQCFIN.
Values:	MQCHT_RECEIVER Receiver. MQCHT_SVRCONN Server-connection (for use by clients). MQCHT_CLUSSDR Cluster-sender.
Returned:	Always.

ErrorIdentifier

Description:	Identifier of the cause of the error. This contains either the reason code (MQRC_* or MQRCCF_*) resulting from the channel definition attempt or the value MQRCCF_SUPPRESSED_BY_EXIT if the attempt to create the definition was disallowed by the exit.

Identifier:	MQIACF_ERROR_IDENTIFIER.
Datatype:	MQCFIN.
Returned:	Always.

ConnectionName

Description:	Name of the partner attempting to establish connection.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Always.

AuxErrorDataInt1

Description:	Auxiliary error data. This contains the value returned by the exit in the *Feedback* field of the MQCXP to indicate why the auto definition has been disallowed.
Identifier:	MQIACF_AUX_ERROR_DATA_INT_1.
Datatype:	MQCFIN.
Returned:	Only if *ErrorIdentifier* contains MQRCCF_SUPPRESSED_BY_EXIT.

Channel Auto-definition OK

Event name:	Channel Auto-definition OK.
Reason code in MQCFH:	MQRC_CHANNEL_AUTO_DEF_OK (2233, X'8B9'). Automatic channel definition succeeded.
Event description:	This condition is detected when the automatic definition of a channel is successful. The channel is defined by the MCA.
Event type:	Channel.
Platforms:	Any MQSeries Version 5 product, except MQSeries for OS/390 when using CICS for distributed queuing.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ChannelName

Description:	Name of the channel being defined.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

ChannelType

Description:	Type of channel being defined.
Identifier:	MQIACH_CHANNEL_TYPE.
Datatype:	MQCFIN.
Values:	MQCHT_RECEIVER Receiver. MQCHT_SVRCONN Server-connection (for use by clients). MQCHT_CLUSSDR Cluster-sender.
Returned:	Always.

ConnectionName

Description:	Name of the partner attempting to establish connection.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Always.

Channel Conversion Error

Event name:	Channel Conversion Error.
Reason code in MQCFH:	MQRC_CHANNEL_CONV_ERROR (2284, X'8EC'). Channel conversion error.
Event description:	This condition is detected when a channel is unable to carry out data conversion and the MQGET call to get a message from the transmission queue resulted in a data conversion error. The reason for the failure is identified by *ConversionReasonCode*.
Event type:	Channel.
Platforms:	All, except MQSeries for OS/390 if CICS is used for distributed queue management.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Note: MQSeries for Windows V2.1 ***does not*** define the channel event queue for you, so the default action is ***not to generate channel events***. This is because, once you have defined a channel event queue, you cannot stop channel event messages being generated. If you want MQSeries to generate channel events, you must define the channel event queue yourself using the name SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ConversionReasonCode

Description:	Identifier of the cause of the conversion error.
Identifier:	MQIACF_CONV_REASON_CODE.
Datatype:	MQCFIN.

Channel Conversion Error

Values:	MQRC_CONVERTED_MSG_TOO_BIG (2120, X'848') Converted message too big for application buffer. MQRC_FORMAT_ERROR (2110, X'83E') Message format not valid. MQRC_NOT_CONVERTED (2119, X'847') Application message data not converted. MQRC_SOURCE_CCSID_ERROR (2111, X'83F') Source coded character set identifier not valid. MQRC_SOURCE_DECIMAL_ENC_ERROR (2113, X'841') Packed-decimal encoding in message not recognized. MQRC_SOURCE_FLOAT_ENC_ERROR (2114, X'842') Floating-point encoding in message not recognized. MQRC_SOURCE_INTEGER_ENC_ERROR (2112, X'840') Integer encoding in message not recognized. MQRC_TARGET_CCSID_ERROR (2115, X'843') Target coded character set identifier not valid. MQRC_TARGET_DECIMAL_ENC_ERROR (2117, X'845') Packed-decimal encoding specified by receiver not recognized. MQRC_TARGET_FLOAT_ENC_ERROR (2118, X'846') Floating-point encoding specified by receiver not recognized. MQRC_TARGET_INTEGER_ENC_ERROR (2116, X'844') Integer encoding specified by receiver not recognized. MQRC_TRUNCATED_MSG_ACCEPTED (2079, X'81F') Truncated message returned (processing completed). MQRC_TRUNCATED_MSG_FAILED (2080, X'820') Truncated message returned (processing not completed).
Returned:	Always.

ChannelName

Description:	Channel name.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

Format

Description:	Format name.
Identifier:	MQCACH_FORMAT_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_FORMAT_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCACH_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.

Returned:	Always.

ConnectionName

Description:	If the channel has successfully established a TCP connection, this is the Internet address. Otherwise it is the contents of the *ConnectionName* field in the channel definition.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Always.

Channel Not Activated

Event name:	Channel Not Activated.
Reason code in MQCFH:	MQRC_CHANNEL_NOT_ACTIVATED (2296, X'8F8'). Channel cannot be activated.
Event description:	This condition is detected when a channel is required to become active, either because it is starting, or because it is about to make another attempt to establish connection with its partner. However, it is unable to do so because the limit on the number of active channels has been reached. See the: • MaxActiveChannels parameter in the qm.ini file for OS/2, AIX, HP-UX, and Sun Solaris • MaxActiveChannels parameter in the Registry for Windows NT • ACTCHL parameter in CSQXPARM for OS/390 The channel waits until it is able to take over an active slot released when another channel ceases to be active. At that time a Channel Activated event is generated.
Event type:	Channel.
Platforms:	All, except MQSeries for OS/390 if CICS is used for distributed queue management.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Note: MQSeries for Windows V2.1 ***does not*** define the channel event queue for you, so the default action is ***not to generate channel events***. This is because, once you have defined a channel event queue, you cannot stop channel event messages being generated. If you want MQSeries to generate channel events, you must define the channel event queue yourself using the name SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ChannelName

Description:	Channel name.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCACH_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.

Returned:	For sender, server, cluster-sender, and cluster-receiver channel types only.

ConnectionName

Description:	If the channel has successfully established a TCP connection, this is the Internet address. Otherwise it is the contents of the *ConnectionName* field in the channel definition.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Only for commands that do not contain a generic name.

Channel Started

Event name:	Channel Started.
Reason code in MQCFH:	MQRC_CHANNEL_STARTED (2282, X'8EA'). Channel started.
Event description:	Either an operator has issued a Start Channel command, or an instance of a channel has been successfully established. This condition is detected when Initial Data negotiation is complete and resynchronization has been performed where necessary, such that message transfer can proceed.
Event type:	Channel.
Platforms:	All, except MQSeries for OS/390 if CICS is used for distributed queue management. Client connections on MQSeries for OS/390, or MQSeries Version 5 products do not produce this event.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Note: MQSeries for Windows V2.1 ***does not*** define the channel event queue for you, so the default action is ***not to generate channel events***. This is because, once you have defined a channel event queue, you cannot stop channel event messages being generated. If you want MQSeries to generate channel events, you must define the channel event queue yourself using the name SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ChannelName

Description:	Channel name.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCACH_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	For sender, server, cluster-sender, and cluster-receiver channels only.

ConnectionName

Description:	If the channel has successfully established a TCP connection, this is the Internet address. Otherwise it is the contents of the *ConnectionName* field in the channel definition.
Identifier:	MQCACH_CONNECTION_NAME.

Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Only for commands that do not contain a generic name.

Channel Stopped

Event name:	Channel Stopped.
Reason code in MQCFH:	MQRC_CHANNEL_STOPPED (2283, X'8EB'). Channel stopped.
Event description:	This condition is detected when a channel has been stopped. *ReasonQualifier* identifies the reasons for stopping.
Event type:	Channel.
Platforms:	All, except MQSeries for OS/390 if CICS is used for distributed queue management. Client connections on MQSeries for OS/390, or MQSeries Version 5 products do not produce this event.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Note: MQSeries for Windows V2.1 ***does not*** define the channel event queue for you, so the default action is ***not to generate channel events***. This is because, once you have defined a channel event queue, you cannot stop channel event messages being generated. If you want MQSeries to generate channel events, you must define the channel event queue yourself using the name SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier that qualifies the reason code.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_CHANNEL_STOPPED_OK Channel has been closed with either a zero return code or a warning return code. MQRQ_CHANNEL_STOPPED_ERROR Channel has been closed but there is an error reported and the channel is not in stopped or retry state. MQRQ_CHANNEL_STOPPED_RETRY Channel has been closed and it is in retry state. MQRQ_CHANNEL_STOPPED_DISABLED Channel has been closed and it is in a stopped state.
Returned:	Always.

ChannelName

Description:	Channel name.
Identifier:	MQCACH_CHANNEL_NAME.

Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

ErrorIdentifier

Description:	Identifier of the cause of the error. If a channel is stopped due to an error, this is the code that identifies the error. If the event message is because of a channel stop failure, the following fields are set: 1. *ReasonQualifier*, containing the value MQRQ_CHANNEL_STOPPED_ERROR 2. *ErrorIdentifier*, containing the code number of an error message that describes the error 3. *AuxErrorDataInt1*, containing error message integer insert 1 4. *AuxErrorDataInt2*, containing error message integer insert 2 5. *AuxErrorDataStr1*, containing error message string insert 1 6. *AuxErrorDataStr2*, containing error message string insert 2 7. *AuxErrorDataStr3*, containing error message string insert 3 The meanings of the error message inserts depend on the code number of the error message. Details of error-message code numbers and the inserts for specific platforms can be found as follows: • For OS/390, see the section “Distributed queuing message codes” in the *MQSeries for OS/390 Messages and Codes* book. • For other platforms, the last four digits of *ErrorIdentifier* when displayed in hexadecimal notation indicate the decimal code number of the error message. For example, if *ErrorIdentifier* has the value X'xxxxyyyy', the message code of the error message explaining the error is AMQyyyy. See the *MQSeries Messages* book for a description of these error messages.
Identifier:	MQIACF_ERROR_IDENTIFIER.
Datatype:	MQCFIN.
Returned:	Always.

AuxErrorDataInt1

Description:	First integer of auxiliary error data for channel errors. If a channel is stopped due to an error, this is the first integer parameter that qualifies the error. This information is for use by IBM service personnel; include it in any problem report that you submit to IBM regarding this event message.
Identifier:	MQIACF_AUX_ERROR_DATA_INT_1.
Datatype:	MQCFIN.
Returned:	Always.

AuxErrorDataInt2

Description:	Second integer of auxiliary error data for channel errors. If a channel is stopped due to an error, this is the second integer parameter that qualifies the error. This information is for use by IBM service personnel; include it in any problem report that you submit to IBM regarding this event message.
Identifier:	MQIACF_AUX_ERROR_DATA_INT_2.
Datatype:	MQCFIN.
Returned:	Always.

AuxErrorDataStr1

Description:	First string of auxiliary error data for channel errors. If a channel is stopped due to an error, this is the first string parameter that qualifies the error. This information is for use by IBM service personnel; include it in any problem report that you submit to IBM regarding this event message.
Identifier:	MQCACF_AUX_ERROR_DATA_STR_1.
Datatype:	MQCFST.
Returned:	Always.

AuxErrorDataStr2

Description:	Second string of auxiliary error data for channel errors. If a channel is stopped due to an error, this is the second string parameter that qualifies the error. This information is for use by IBM service personnel; include it in any problem report that you submit to IBM regarding this event message.
Identifier:	MQCACF_AUX_ERROR_DATA_STR_2.
Datatype:	MQCFST.
Returned:	Always.

AuxErrorDataStr3

Description:	Third string of auxiliary error data for channel errors. If a channel is stopped due to an error, this is the third string parameter that qualifies the error. This information is for use by IBM service personnel; include it in any problem report that you submit to IBM regarding this event message.
Identifier:	MQCACF_AUX_ERROR_DATA_STR_3.
Datatype:	MQCFST.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCACH_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	For sender, server, cluster-sender, and cluster-receiver channels only.

ConnectionName

Description:	If the channel has successfully established a TCP connection, this is the Internet address. Otherwise it is the contents of the *ConnectionName* field in the channel definition.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Only for commands that do not contain a generic name.

Channel Stopped By User

Event name:	Channel Stopped By User.
Reason code in MQCFH:	MQRC_CHANNEL_STOPPED_BY_USER (2279, X'8E7'). Channel stopped by user.
Event description:	This condition is detected when a channel has been stopped by the operator. *ReasonQualifier* identifies the reasons for stopping.
Event type:	Channel.
Platforms:	All, except MQSeries for Tandem NonStop Kernel, MQSeries for Compaq Tru64 UNIX, or MQSeries for OS/390 if CICS is used for distributed queue management.
Event queue:	SYSTEM.ADMIN.CHANNEL.EVENT.

Note: MQSeries for Windows V2.1 ***does not*** define the channel event queue for you, so the default action is ***not to generate channel events***. This is because, once you have defined a channel event queue, you cannot stop channel event messages being generated. If you want MQSeries to generate channel events, you must define the channel event queue yourself using the name SYSTEM.ADMIN.CHANNEL.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier that qualifies the reason code.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_CHANNEL_STOPPED_DISABLED Channel has been closed and it is in a stopped state.
Returned:	Always.

ChannelName

Description:	Channel name.
Identifier:	MQCACH_CHANNEL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CHANNEL_NAME_LENGTH.
Returned:	Always.

Channel Stopped By User

ErrorIdentifier

Description:	Identifier of the cause of the error. As the event message is generated by a Stop Channel command and not a channel error, the following fields are set: 1. *ReasonQualifier*, containing the same value as in the *ReasonQualifier*(MQCFIN) field. 2. *AuxErrorDataInt1*, containing zeros 3. *AuxErrorDataInt2*, containing zeros 4. *AuxErrorDataStr1*, containing zeros 5. *AuxErrorDataStr2*, containing zeros 6. *AuxErrorDataStr3*, containing zeros
Identifier:	MQIACF_ERROR_IDENTIFIER.
Datatype:	MQCFIN.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCACH_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	For sender, server, cluster-sender, and cluster-receiver channels only.

ConnectionName

Description:	If the channel has successfully established a TCP connection, this is the Internet address. Otherwise it is the contents of the *ConnectionName* field in the channel definition.
Identifier:	MQCACH_CONNECTION_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_CONN_NAME_LENGTH.
Returned:	Only for commands that do not contain a generic name.

Default Transmission Queue Type Error

Event name:	Default Transmission Queue Type Error.
Reason code in MQCFH:	MQRC_DEF_XMIT_Q_TYPE_ERROR (2198, X'896'). Default transmission queue not local.
Event description:	An MQOPEN or MQPUT1 call was issued specifying a remote queue as the destination. Either a local definition of the remote queue was specified, or a queue-manager alias was being resolved, but in either case the *XmitQName* attribute in the local definition is blank. No transmission queue is defined with the same name as the destination queue manager, so the local queue manager has attempted to use the default transmission queue. However, although there is a queue defined by the *DefXmitQName* queue-manager attribute, it is not a local queue. See the *MQSeries Application Programming Guide* for more information.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Default transmission queue name.
Identifier:	MQCA_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

QType

Description:	Type of default transmission queue.
Identifier:	MQIA_Q_TYPE.
Datatype:	MQCFIN.

Values:	MQQT_ALIAS Alias queue definition. MQQT_REMOTE Local definition of a remote queue.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Default Transmission Queue Usage Error

Event name:	Default Transmission Queue Usage Error.
Reason code in MQCFH:	MQRC_DEF_XMIT_Q_USAGE_ERROR (2199, X'897'). Default transmission queue usage error.
Event description:	An MQOPEN or MQPUT1 call was issued specifying a remote queue as the destination. Either a local definition of the remote queue was specified, or a queue-manager alias was being resolved, but in either case the *XmitQName* attribute in the local definition is blank. No transmission queue is defined with the same name as the destination queue manager, so the local queue manager has attempted to use the default transmission queue. However, the queue defined by the *DefXmitQName* queue-manager attribute does not have a *Usage* attribute of MQUS_TRANSMISSION. See the *MQSeries Application Programming Guide* for more information.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Default transmission queue name.
Identifier:	MQCA_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

Default Transmission Queue Usage Error

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Get Inhibited

Event name:	Get Inhibited.
Reason code in MQCFH:	MQRC_GET_INHIBITED (2016, X'7E0'). Gets inhibited for the queue.
Event description:	MQGET calls are currently inhibited for the queue (see the *InhibitGet* queue attribute in the *MQSeries Application Programming Reference* manual) or for the queue to which this queue resolves.
Event type:	Inhibit.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application that issued the get.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application that issued the get.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Not Authorized (type 1)

Event name:	Not Authorized (type 1).
Reason code in MQCFH:	MQRC_NOT_AUTHORIZED (2035, X'7F3'). Not authorized for access.
Event description:	On an MQCONN call, the user is not authorized to connect to the queue manager.
Event type:	Authority.
Platforms:	All, except MQSeries for OS/390, MQSeries for OS/2 Warp, and MQSeries for Windows Version 2.1.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier for type 1 authority events.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_CONN_NOT_AUTHORIZED Connection not authorized.
Returned:	Always.

UserIdentifier

Description:	User identifier that caused the authorization check.
Identifier:	MQCACF_USER_IDENTIFIER.
Datatype:	MQCFST.
Maximum length:	MQ_USER_ID_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application causing the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application causing the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Not Authorized (type 2)

Event name:	Not Authorized (type 2).
Reason code in MQCFH:	MQRC_NOT_AUTHORIZED (2035, X'7F3'). Not authorized for access.
Event description:	On an MQOPEN or MQPUT1 call, the user is not authorized to open the object for the options specified.
Event type:	Authority.
Platforms:	All, except MQSeries for OS/390, MQSeries for OS/2 Warp, MQSeries for Tandem NSK, and MQSeries for Windows Version 2.1.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier for type 2 authority events.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_OPEN_NOT_AUTHORIZED Open not authorized.
Returned:	Always.

Options

Description:	Options specified on the MQOPEN call.
Identifier:	MQIACF_OPEN_OPTIONS.
Datatype:	MQCFIN.
Returned:	Always.

UserIdentifier

Description:	User identifier that caused the authorization check.
Identifier:	MQCACF_USER_IDENTIFIER.
Datatype:	MQCFST.
Maximum length:	MQ_USER_ID_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application that caused the authorization check.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application that caused the authorization check.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Object queue manager name from object descriptor (MQOD).
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectQMgrName* in the object descriptor (MQOD) when the object was opened is not the queue manager currently connected.

QName

Description:	Object name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	If the object opened is not a process object.

ProcessName

Description:	Name of process object from object descriptor (MQOD).
Identifier:	MQCA_PROCESS_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_PROCESS_NAME_LENGTH.
Returned:	If the object opened is a process object.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Not Authorized (type 3)

Event name:	Not Authorized (type 3).
Reason code in MQCFH:	MQRC_NOT_AUTHORIZED (2035, X'7F3'). Not authorized for access.
Event description:	On an MQCLOSE call, the user is not authorized to delete the object, which is a permanent dynamic queue, and the *Hobj* parameter specified on the MQCLOSE call is not the handle returned by the MQOPEN call that created the queue.
Event type:	Authority.
Platforms:	All, except MQSeries for OS/390, MQSeries for OS/2 Warp, MQSeries for Tandem NSK, and MQSeries for Windows Version 2.1.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier for type 3 authority events.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_CLOSE_NOT_AUTHORIZED Close not authorized.
Returned:	Always.

QName

Description:	Object name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

UserIdentifier

Description:	User identifier that caused the authorization check.
Identifier:	MQCACF_USER_IDENTIFIER.
Datatype:	MQCFST.
Maximum length:	MQ_USER_ID_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application causing the authorization check.
Identifier:	MQIA_APPL_TYPE.

Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application causing the authorization check.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Not Authorized (type 4)

Event name:	Not Authorized (type 4).
Reason code in MQCFH:	MQRC_NOT_AUTHORIZED (2035, X'7F3'). Not authorized for access.
Event description:	Indicates that a command has been issued from a user ID that is not authorized to access the object specified in the command.
Event type:	Authority.
Platforms:	All, except MQSeries for OS/390, MQSeries for OS/2 Warp, MQSeries for Tandem NSK, and MQSeries for Windows Version 2.1.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier for type 4 authority events.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_CMD_NOT_AUTHORIZED Command not authorized.
Returned:	Always.

Command

Description:	Command identifier. See the MQCFH header structure, described in “MQCFH (Event header)” on page 41.
Identifier:	MQIACF_COMMAND.
Datatype:	MQCFIN.
Returned:	Always.

UserIdentifier

Description:	User identifier that caused the authorization check.
Identifier:	MQCACF_USER_IDENTIFIER.
Datatype:	MQCFST.
Maximum length:	MQ_USER_ID_LENGTH.
Returned:	Always.

Put Inhibited

Event name:	Put Inhibited.
Reason code in MQCFH:	MQRC_PUT_INHIBITED (2051, X'803'). Put calls inhibited for the queue.
Event description:	MQPUT and MQPUT1 calls are currently inhibited for the queue (see the *InhibitPut* queue attribute in in the *MQSeries Application Programming Reference* manual) or for the queue to which this queue resolves.
Event type:	Inhibit.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application that issued the put.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application that issued the put.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of queue manager from object descriptor (MQOD).
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.

Put Inhibited

Returned:	Only if this parameter has a value different from *QMgrName*. This occurs when the *ObjectQMgrName* field in the object descriptor provided by the application on the MQOPEN or MQPUT1 call is neither blank nor the name of the application's local queue manager. However, it can also occur when *ObjectQMgrName* in the object descriptor is blank, but a name service provides a queue-manager name that is not the name of the application's local queue manager.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Queue Depth High

Event name:	Queue Depth High.
Reason code in MQCFH:	MQRC_Q_DEPTH_HIGH (2224, X'8B0'). Queue depth high limit reached or exceeded.
Event description:	An MQPUT or MQPUT1 call has caused the queue depth to be incremented to or above the limit specified in the *QDepthHighLimit* attribute.
Corrective action:	None. This reason code is used only to identify the corresponding event message.
Event type:	Performance.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.PERFM.EVENT.

Notes:

1. MQSeries for OS/390 supports queue depth events on shared queues. You might receive a NULL event message for a shared queue if a queue manager has performed no activity on that shared queue.
2. For shared queues, the correlation identifier, *CorrelId* in the message descriptor (MQMD) is set. See "MQMD (Message descriptor)" on page 35 for more information.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Name of the queue on which the limit has been reached.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

TimeSinceReset

Description:	Time, in seconds, since the statistics were last reset. The value recorded by this timer is also used as the *interval time* in queue service interval events.
Identifier:	MQIA_TIME_SINCE_RESET.
Datatype:	MQCFIN.
Returned:	Always.

HighQDepth

Description:	Maximum number of messages on the queue since the queue statistics were last reset.
Identifier:	MQIA_HIGH_Q_DEPTH.
Datatype:	MQCFIN.

Returned:	Always.

MsgEnqCount

Description:	Number of messages enqueued. This is the number of messages put on the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_ENQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

MsgDeqCount

Description:	Number of messages removed from the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_DEQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

Queue Depth Low

Event name:	Queue Depth Low.
Reason code in MQCFH:	MQRC_Q_DEPTH_LOW (2225, X'8B1'). Queue depth low limit reached or exceeded.
Event description:	An MQGET call has caused the queue depth to be decremented to or below the limit specified in the *QDepthLowLimit* attribute.
Corrective action:	None. This reason code is used only to identify the corresponding event message.
Event type:	Performance.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.PERFM.EVENT.

Notes:

1. MQSeries for OS/390 supports queue depth events on shared queues. You might receive a NULL event message for a shared queue if a queue manager has performed no activity on that shared queue.
2. For shared queues, the correlation identifier, *CorrelId* in the message descriptor (MQMD) is set. See "MQMD (Message descriptor)" on page 35 for more information.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Name of the queue on which the limit has been reached.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

TimeSinceReset

Description:	Time, in seconds, since the statistics were last reset. The value recorded by this timer is also used as the *interval time* in queue service interval events.
Identifier:	MQIA_TIME_SINCE_RESET.
Datatype:	MQCFIN.
Returned:	Always.

HighQDepth

Description:	Maximum number of messages on the queue since the queue statistics were last reset.
Identifier:	MQIA_HIGH_Q_DEPTH.
Datatype:	MQCFIN.

Returned:	Always.

MsgEnqCount

Description:	Number of messages enqueued. This is the number of messages put on the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_ENQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

MsgDeqCount

Description:	Number of messages removed from the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_DEQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

Queue Full

Event name:	Queue Full.
Reason code in MQCFH:	MQRC_Q_FULL (2053, X'805'). Queue already contains maximum number of messages.
Event description:	On an MQPUT or MQPUT1 call, the call failed because the queue is full. That is, it already contains the maximum number of messages possible (see the *MaxQDepth* local-queue attribute in the *MQSeries Application Programming Reference* manual). This reason code can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.
Corrective action:	Retry the operation later. Consider increasing the maximum depth for this queue, or arranging for more instances of the application to service the queue.
Event type:	Performance.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.PERFM.EVENT.

Notes:

1. MQSeries for OS/390 supports queue depth events on shared queues. You might receive a NULL event message for a shared queue if a queue manager has performed no activity on that shared queue.
2. For shared queues, the correlation identifier, *CorrelId* in the message descriptor (MQMD) is set. See “MQMD (Message descriptor)” on page 35 for more information.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Name of the queue on which the put was rejected.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

TimeSinceReset

Description:	Time, in seconds, since the statistics were last reset.
Identifier:	MQIA_TIME_SINCE_RESET.
Datatype:	MQCFIN.
Returned:	Always.

Queue Full

HighQDepth

Description:	Maximum number of messages on a queue.
Identifier:	MQIA_HIGH_Q_DEPTH.
Datatype:	MQCFIN.
Returned:	Always.

MsgEnqCount

Description:	Number of messages enqueued. This is the number of messages put on the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_ENQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

MsgDeqCount

Description:	Number of messages removed from the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_DEQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

Queue Manager Active

Event name:	Queue Manager Active.
Reason code in MQCFH:	MQRC_Q_MGR_ACTIVE (2222, X'8AE'). Queue manager created.
Event description:	This condition is detected when a queue manager becomes active.
Event type:	Start And Stop.
Platforms:	All, except the first start of an MQSeries for OS/390 queue manager. In this case it is produced only on subsequent restarts.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

Queue Manager Not Active

Event name:	Queue Manager Not Active.
Reason code in MQCFH:	MQRC_Q_MGR_NOT_ACTIVE (2223, X'8AF'). Queue manager unavailable.
Event description:	This condition is detected when a queue manager is requested to stop or quiesce.
Event type:	Start And Stop.
Platforms:	All, except MQSeries for OS/390.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ReasonQualifier

Description:	Identifier of causes of this reason code. This specifies the type of stop that was requested.
Identifier:	MQIACF_REASON_QUALIFIER.
Datatype:	MQCFIN.
Values:	MQRQ_Q_MGR_STOPPING Queue manager stopping. MQRQ_Q_MGR_QUIESCING Queue manager quiescing.
Returned:	Always.

Queue Service Interval High

Event name:	Queue Service Interval High.
Reason code in MQCFH:	MQRC_Q_SERVICE_INTERVAL_HIGH (2226, X'8B2'). Queue service interval high.
Event description:	No successful gets or puts have been detected within an interval greater than the limit specified in the *QServiceInterval* attribute.
Corrective action:	None. This reason code is used only to identify the corresponding event message.
Event type:	Performance.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.PERFM.EVENT.

Note: MQSeries for OS/390 does not support service interval events on shared queues

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Name of the queue specified on the command that caused this queue service interval event to be generated.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

TimeSinceReset

Description:	Time, in seconds, since the statistics were last reset. For a service interval high event, this value is greater than the service interval.
Identifier:	MQIA_TIME_SINCE_RESET.
Datatype:	MQCFIN.
Returned:	Always.

HighQDepth

Description:	Maximum number of messages on the queue since the queue statistics were last reset.
Identifier:	MQIA_HIGH_Q_DEPTH.
Datatype:	MQCFIN.
Returned:	Always.

Queue Service Interval High

MsgEnqCount

Description:	Number of messages enqueued. This is the number of messages put on the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_ENQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

MsgDeqCount

Description:	Number of messages removed from the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_DEQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

Queue Service Interval OK

Event name:	Queue Service Interval OK.
Reason code in MQCFH:	MQRC_Q_SERVICE_INTERVAL_OK (2227, X'8B3'). Queue service interval OK.
Event description:	A successful get has been detected within an interval less than or equal to the limit specified in the *QServiceInterval* attribute.
Corrective action:	None. This reason code is used only to identify the corresponding event message.
Event type:	Performance.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.PERFM.EVENT.

Note: MQSeries for OS/390 does not support service interval events on shared queues.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name specified on the command that caused this queue service interval event to be generated.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

TimeSinceReset

Description:	Time, in seconds, since the statistics were last reset.
Identifier:	MQIA_TIME_SINCE_RESET.
Datatype:	MQCFIN.
Returned:	Always.

HighQDepth

Description:	Maximum number of messages on the queue since the queue statistics were last reset.
Identifier:	MQIA_HIGH_Q_DEPTH.
Datatype:	MQCFIN.
Returned:	Always.

MsgEnqCount

Description:	Number of messages enqueued. This is the number of messages put on the queue since the queue statistics were last reset.

Queue Service Interval OK

Identifier:	MQIA_MSG_ENQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

MsgDeqCount

Description:	Number of messages removed from the queue since the queue statistics were last reset.
Identifier:	MQIA_MSG_DEQ_COUNT.
Datatype:	MQCFIN.
Returned:	Always.

Queue Type Error

Event name:	Queue Type Error.
Reason code in MQCFH:	MQRC_Q_TYPE_ERROR (2057, X'809'). Queue type not valid.
Event description:	On an MQOPEN call, the *ObjectQMgrName* field in the object descriptor specifies the name of a local definition of a remote queue (in order to specify a queue-manager alias). In that local definition the *RemoteQMgrName* attribute is the name of the local queue manager. However, the *ObjectName* field specifies the name of a model queue on the local queue manager, which is not allowed. See the *MQSeries Application Programming Guide* for more information.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.

Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Remote Queue Name Error

Event name:	Remote Queue Name Error.
Reason code in MQCFH:	MQRC_REMOTE_Q_NAME_ERROR (2184, X'888'). Remote queue name not valid.
Event description:	On an MQOPEN or MQPUT1 call either: • A local definition of a remote queue (or an alias to one) was specified, but the *RemoteQName* attribute in the remote queue definition is blank. Note that this error occurs even if the *XmitQName* in the definition is not blank. or • the *ObjectQMgrName* field in the object descriptor was not blank and not the name of the local queue manager, but the *ObjectName* field is blank.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.

Remote Queue Name Error

Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Transmission Queue Type Error

Event name:	Transmission Queue Type Error.
Reason code in MQCFH:	MQRC_XMIT_Q_TYPE_ERROR (2091, X'82B'). Transmission queue not local.
Event description:	On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. The *ObjectName* or *ObjectQMgrName* field in the object descriptor specifies the name of a local definition of a remote queue but one of the following applies to the *XmitQName* attribute of the definition. Either: • *XmitQName* is not blank, but specifies a queue that is not a local queue or • *XmitQName* is blank, but *RemoteQMgrName* specifies a queue that is not a local queue This also occurs if the queue name is resolved through a cell directory, and the remote queue manager name obtained from the cell directory is the name of a queue, but this is not a local queue.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCA_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

QType

Description:	Type of transmission queue.
Identifier:	MQIA_Q_TYPE.
Datatype:	MQCFIN.

Values:	MQQT_ALIAS Alias queue definition. MQQT_REMOTE Local definition of a remote queue.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Transmission Queue Usage Error

Event name:	Transmission Queue Usage Error.
Reason code in MQCFH:	MQRC_XMIT_Q_USAGE_ERROR (2092, X'82C'). Transmission queue with wrong usage.
Event description:	On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager, but one of the following occurred. Either: • *ObjectQMgrName* specifies the name of a local queue, but it does not have a *Usage* attribute of MQUS_TRANSMISSION. or • The *ObjectName* or *ObjectQMgrName* field in the object descriptor specifies the name of a local definition of a remote queue but one of the following applies to the *XmitQName* attribute of the definition: – *XmitQName* is not blank, but specifies a queue that does not have a *Usage* attribute of MQUS_TRANSMISSION – *XmitQName* is blank, but *RemoteQMgrName* specifies a queue that does not have a *Usage* attribute of MQUS_TRANSMISSION or • The queue name is resolved through a cell directory, and the remote queue manager name obtained from the cell directory is the name of a local queue, but it does not have a *Usage* attribute of MQUS_TRANSMISSION.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCA_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Unknown Alias Base Queue

Event name:	Unknown Alias Base Queue.
Reason code in MQCFH:	MQRC_UNKOWN_ALIAS_BASE_Q (2082, X'822'). Unknown alias base queue.
Event description:	An MQOPEN or MQPUT1 call was issued specifying an alias queue as the destination, but the *BaseQName* in the alias queue attributes is not recognized as a queue name.
Event type:	Local.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

BaseQName

Description:	Queue name to which the alias resolves.
Identifier:	MQCA_BASE_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of the application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

Unknown Alias Base Queue

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Unknown Default Transmission Queue

Event name:	Unknown Default Transmission Queue.
Reason code in MQCFH:	MQRC_UNKNOWN_DEF_XMIT_Q (2197, X'895'). Unknown default transmission queue.
Event description:	An MQOPEN or MQPUT1 call was issued specifying a remote queue as the destination. If a local definition of the remote queue was specified, or if a queue-manager alias is being resolved, the *XmitQName* attribute in the local definition is blank. No queue is defined with the same name as the destination queue manager. The queue manager has therefore attempted to use the default transmission queue. However, the name defined by the *DefXmitQName* queue-manager attribute is not the name of a locally-defined queue.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Default transmission queue name.
Identifier:	MQCA_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application attempting to open the remote queue.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application attempting to open the remote queue.

Unknown Default Transmission Queue

Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Unknown Object Name

Event name:	Unknown Object Name.
Reason code in MQCFH:	MQRC_UNKNOWN_OBJECT_NAME (2085, X'825'). Unknown object name.
Event description:	On an MQOPEN or MQPUT1 call, the *ObjectQMgrName* field in the object descriptor MQOD is set to one of the following. It is either: • Blank or • The name of the local queue manager or • The name of a local definition of a remote queue (a queue-manager alias) in which the *RemoteQMgrName* attribute is the name of the local queue manager However, the *ObjectName* in the object descriptor is not recognized for the specified object type. See also MQRC_Q_DELETED.
Event type:	Local.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of the application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.

Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always, unless *ProcessName* is returned. Either *QName* or *ProcessName* is returned.

ProcessName

Description:	Name of the process (application) making the MQI call that caused the event.
Identifier:	MQCA_PROCESS_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_PROCESS_NAME_LENGTH.
Returned:	Always, unless *QName* is returned. Either *ProcessName* or *QName* is returned.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Unknown Remote Queue Manager

Event name:	Unknown Remote Queue Manager.
Reason code in MQCFH:	MQRC_UNKNOWN_REMOTE_Q_MGR (2087, X'827'). Unknown remote queue manager.
Event description:	On an MQOPEN or MQPUT1 call, an error occurred with the queue-name resolution, for one of the following reasons: • *ObjectQMgrName* is either blank or the name of the local queue manager, and *ObjectName* is the name of a local definition of a remote queue that has a blank *XmitQName*. However, there is no (transmission) queue defined with the name of *RemoteQMgrName*, and the *DefXmitQName* queue-manager attribute is blank. • *ObjectQMgrName* is the name of a queue-manager alias definition (held as the local definition of a remote queue) that has a blank *XmitQName*. However, there is no (transmission) queue defined with the name of *RemoteQMgrName*, and the *DefXmitQName* queue-manager attribute is blank. • *ObjectQMgrName* specified is not: – Blank – The name of the local queue manager – The name of a local queue – The name of a queue-manager alias definition (that is, a local definition of a remote queue with a blank *RemoteQName*) and the *DefXmitQName* queue-manager attribute is blank. • *ObjectQMgrName* is blank or is the name of the local queue manager, and *ObjectName* is the name of a local definition of a remote queue (or an alias to one), for which *RemoteQMgrName* is either blank or is the name of the local queue manager. Note that this error occurs even if the *XmitQName* is not blank. • *ObjectQMgrName* is the name of a local definition of a remote queue. In this case, it should be a queue-manager alias definition, but the *RemoteQName* in the definition is not blank. • *ObjectQMgrName* is the name of a model queue. • The queue name is resolved through a cell directory. However, there is no queue defined with the same name as the remote queue manager name obtained from the cell directory. Also, the *DefXmitQName* queue-manager attribute is blank.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.

Returned:	Always.

ApplType

Description:	Type of application attempting to open the remote queue.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application attempting to open the remote queue.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.
Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Unknown Transmission Queue

Event name:	Unknown Transmission Queue.
Reason code in MQCFH:	MQRC_UNKNOWN_XMIT_Q (2196, X'894'). Unknown transmission queue.
Event description:	On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. The *ObjectName* or the *ObjectQMgrName* in the object descriptor specifies the name of a local definition of a remote queue (in the latter case queue-manager aliasing is being used). However, the *XmitQName* attribute of the definition is not blank and not the name of a locally-defined queue.
Event type:	Remote.
Platforms:	All.
Event queue:	SYSTEM.ADMIN.QMGR.EVENT.

Event data

QMgrName

Description:	Name of the queue manager generating the event.
Identifier:	MQCA_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	Always.

QName

Description:	Queue name from object descriptor (MQOD).
Identifier:	MQCA_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

XmitQName

Description:	Transmission queue name.
Identifier:	MQCA_XMIT_Q_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_NAME_LENGTH.
Returned:	Always.

ApplType

Description:	Type of application making the MQI call that caused the event.
Identifier:	MQIA_APPL_TYPE.
Datatype:	MQCFIN.
Returned:	Always.

ApplName

Description:	Name of the application making the MQI call that caused the event.
Identifier:	MQCACF_APPL_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_APPL_NAME_LENGTH.

Returned:	Always.

ObjectQMgrName

Description:	Name of the object queue manager.
Identifier:	MQCACF_OBJECT_Q_MGR_NAME.
Datatype:	MQCFST.
Maximum length:	MQ_Q_MGR_NAME_LENGTH.
Returned:	If the *ObjectName* in the object descriptor (MQOD), when the object was opened, is not the queue manager currently connected.

Note: If the application is a server for clients, the *ApplType* and *ApplName* parameters identify the server, not the client.

Chapter 4. Example of using instrumentation events

This example shows how you can write a program for instrumentation events. It is written in C for queue managers on OS/2 Warp, Windows NT, AS/400, or UNIX systems. It is not part of any MQSeries product and is therefore supplied as source only. The example is incomplete in that it does not enumerate all the possible outcomes of specified actions. Bearing this in mind, 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 will need to modify this program to get it to run on your 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
#ifdef OS2
  /****************************************************************/
  /* for beep                                                     */
  /****************************************************************/
  #define INCL_DOSPROCESS
  #include <os2.h>
#endif
```

Figure 10. Event monitoring sample program (Part 1 of 11)

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

Figure 10. Event monitoring sample program (Part 2 of 11)

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

Figure 10. Event monitoring sample program (Part 3 of 11)

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

Figure 10. Event monitoring sample program (Part 4 of 11)

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

Figure 10. Event monitoring sample program (Part 5 of 11)

```
if (CompCode != MQCC_FAILED)
{
  /************************************************************/
  /* announce a message                                       */
  /************************************************************/
  #ifdef OS2
    {
      unsigned short tone;
      for (tone = 1; tone < 8000; tone = tone * 2)
      {
        DosBeep(tone,50);
      }
    }
  #else
    printf("\a\a\a\a\a\a\a");
  #endif
  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)
    {
      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
```

⋮

Figure 10. Event monitoring sample program (Part 6 of 11)

```
    {
      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             */
  /************************************************************/
```

Figure 10. Event monitoring sample program (Part 7 of 11)

```
        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);
  }     /* 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                  */
```

Figure 10. Event monitoring sample program (Part 8 of 11)

```
 /********************************************************************/
 /*                                                                  */
 /* 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;
  }
}
```

Figure 10. Event monitoring sample program (Part 9 of 11)

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

Figure 10. Event monitoring sample program (Part 10 of 11)

```
          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;
  }
}
```

Figure 10. Event monitoring sample program (Part 11 of 11)

Appendix A. Structure datatypes MQCFIN and MQCFST

In this appendix, the structures MQCFIN and MQCFST are described in a language-independent form. The declarations are shown in the following programming languages:
- C
- COBOL
- PL/I (AIX, OS/2 Warp, OS/390, and Windows NT)
- S/390® assembler (OS/390 only)
- Visual Basic (Windows platforms only)

The elementary data types of the fields in MQCFIN and MQCFST are described in the *MQSeries Application Programming Reference* manual.

The *initial value* of each field is shown under its description. This is the value of the field in the *default structure*.

MQCFIN - Integer parameter

The MQCFIN structure describes an integer parameter in an event message.

Type

Description:	Indicates that the structure type is MQCFIN and describes an integer parameter.
Datatype:	MQLONG.
Initial value:	MQCFT_INTEGER.
Valid value:	MQCFT_INTEGER Structure defining an integer.

StrucLength

Description:	Length in bytes of the MQCFIN structure.
Datatype:	MQLONG.
Initial value:	MQCFIN_STRUC_LENGTH.
Valid value:	MQCFIN_STRUC_LENGTH Length of MQCFIN structure.

Parameter

Description:	Identifies the parameter whose value is contained in the structure.
Datatype:	MQLONG.
Initial value:	0.
Valid values:	Dependent on the event message.

Value

Description:	Value of parameter identified by the *Parameter* field.
Datatype:	MQLONG.
Initial value:	0.

C language declaration (MQCFIN)

```
typedef struct tagMQCFIN {
  MQLONG  Type;         /* Structure type */
  MQLONG  StrucLength;  /* Structure length */
  MQLONG  Parameter;    /* Parameter identifier */
  MQLONG  Value;        /* Parameter value */
 } MQCFIN;
```

COBOL language declaration (MQCFIN)

```
**   MQCFIN structure
  10 MQCFIN.
**    Structure type
   15 MQCFIN-TYPE        PIC S9(9) BINARY.
**    Structure length
   15 MQCFIN-STRUCLENGTH PIC S9(9) BINARY.
**    Parameter identifier
   15 MQCFIN-PARAMETER   PIC S9(9) BINARY.
**    Parameter value
   15 MQCFIN-VALUE       PIC S9(9) BINARY.
```

PL/I language declaration (MQCFIN)

```
dcl
 1 MQCFIN based,
  3 Type        fixed bin(31), /* Structure type */
  3 StrucLength fixed bin(31), /* Structure length */
  3 Parameter   fixed bin(31), /* Parameter identifier */
  3 Value       fixed bin(31); /* Parameter value */
```

System/390 assembler-language declaration (MQCFIN)

```
MQCFIN                         DSECT
MQCFIN_TYPE                    DS   F        Structure type
MQCFIN_STRUCLENGTH             DS   F        Structure length
MQCFIN_PARAMETER               DS   F        Parameter identifier
MQCFIN_VALUE                   DS   F        Parameter value
MQCFIN_LENGTH                  EQU  *-MQCFIN Length of structure
                               ORG  MQCFIN
MQCFIN_AREA                    DS   CL(MQCFIN_LENGTH)
```

Visual Basic language declaration (MQCFIN)

```
Type MQCFIN
  Type As Long         ' Structure type
  StrucLength As Long  ' Structure length
  Parameter As Long    ' Parameter identifier
  Value As Long        ' Parameter value
End Type

Global MQCFIN_DEFAULT As MQCFIN
```

MQCFST - String parameter

The MQCFST structure describes a string parameter in an event message.

The structure ends with a variable-length character string; see the *String* field below for further details.

Type

Description:	Indicates that the structure type is MQCFST and describes a string parameter.

Datatype:	MQLONG.
Initial value:	MQCFT_STRING.
Valid value:	MQCFT_STRING Structure defining a string.

StrucLength

Description:	Length in bytes of the MQCFST structure, including the string at the end of the structure (the *String* field). The length must be a multiple of four, and must be sufficient to contain the string; any bytes between the end of the string and the length defined by the *StrucLength* field are not significant.
Datatype:	MQLONG.
Initial value:	MQCFST_STRUC_LENGTH_FIXED. Length of the *fixed* part of the MQCFST structure, excluding the *String* field.

Parameter

Description:	Identifies the parameter whose value is contained in the structure.
Datatype:	MQLONG.
Initial value:	0.
Valid values:	Dependent on the event message.

CodedCharSetId

Description: Coded character set identifier of the data in the *String* field.

Datatype: MQLONG.

Initial value:

MQCCSI_DEFAULT.
: Default coded character set identifier, indicating that character data is in the character set defined by the *CodedCharSetId* field in the MQ header structure that **precedes** the MQCFH structure, or by the *CodedCharSetId* field in the MQMD if the MQCFH structure is at the start of the message.

Valid values:

- If all of the strings in an event message have the same coded character-set identifier, the *CodedCharSetId* field in the message descriptor MQMD or in the MQ header structure preceding MQCFH should be set to that identifier when the message is put, and the *CodedCharSetId* fields in the MQCFST structure within the message should be set to MQCCSI_DEFAULT.
- If some of the strings in the message have different character-set identifiers, the *CodedCharSetId* field in MQMD or in the MQ header structure preceding MQCFH should be set to MQCCSI_EMBEDDED when the message is put, and the *CodedCharSetId* fields in the MQCFST structure within the message should be set to the identifiers that apply.

Do not specify MQCCSI_EMBEDDED in MQMD or in the MQ header structure preceding MQCFH when the message is put, with MQCCSI_DEFAULT in the MQCFST structure within the message, as this will prevent conversion of the message.

StringLength

Description:	Length in bytes of the data in the *String* field; it must be zero or greater. This length need not be a multiple of four.
Datatype:	MQLONG.
Initial value:	0.

String

Description:	The value of the parameter identified by the *Parameter* field. In MQFMT_EVENT messages, trailing blanks are omitted from string parameters (that is, the string may be shorter than the defined length of the parameter). *StringLength* gives the length of the string actually present in the message.
Datatype:	MQCHAR×*StringLength*.
Initial value:	In C, the initial value of this field is the null string.
Valid value:	The string can contain any characters that are in the character set defined by *CodedCharSetId*, and that are valid for the parameter identified by *Parameter*.
Language considerations:	The way that this field is declared depends on the programming language: • For the C programming language, the field is declared as an array with one element. Storage for the structure should be allocated dynamically, and pointers used to address the fields within it. • For the COBOL, PL/I, System/390 assembler, and Visual Basic programming languages, the field is omitted from the structure declaration. When an instance of the structure is declared, the user should include MQCFST in a larger structure, and declare additional fields following MQCFST, to represent the *String* field as required.
Special note:	A null character in the string is treated as normal data, and does not act as a delimiter for the string. This means that when a receiving application reads an MQFMT_EVENT message, the receiving application receives all of the data specified by the sending application. The data may, of course, have been converted between character sets (for example, by the receiving application specifying the MQGMO_CONVERT option on the MQGET call).

C language declaration (MQCFST)

```
typedef struct tagMQCFST {
  MQLONG  Type;            /* Structure type */
  MQLONG  StrucLength;     /* Structure length */
  MQLONG  Parameter;       /* Parameter identifier */
  MQLONG  CodedCharSetId;  /* Coded character set identifier */
  MQLONG  StringLength;    /* Length of string */
  MQCHAR  String[1];       /* String value - first
                              character */
 } MQCFST;
```

In the C programming language, the macro variable MQCFST_DEFAULT contains the initial values of the MQCFST structure. It can be used in the following way to provide initial values for the fields in the structure:

```
struct {
  MQCFST Hdr;
  MQCHAR Data[99];
} MyCFST = {MQCFST_DEFAULT};
```

COBOL language declaration (MQCFST)

```
**   MQCFST structure
  10 MQCFST.
**    Structure type
   15 MQCFST-TYPE           PIC S9(9) BINARY.
**    Structure length
   15 MQCFST-STRUCLENGTH    PIC S9(9) BINARY.
**    Parameter identifier
   15 MQCFST-PARAMETER      PIC S9(9) BINARY.
**    Coded character set identifier
   15 MQCFST-CODEDCHARSETID PIC S9(9) BINARY.
**    Length of string
   15 MQCFST-STRINGLENGTH   PIC S9(9) BINARY.
```

PL/I language declaration (MQCFST)

```
dcl
 1 MQCFST based,
  3 Type           fixed bin(31), /* Structure type */
  3 StrucLength    fixed bin(31), /* Structure length */
  3 Parameter      fixed bin(31), /* Parameter identifier */
  3 CodedCharSetId fixed bin(31), /* Coded character set identifier */
  3 StringLength   fixed bin(31); /* Length of string */
```

System/390 assembler-language declaration (MQCFST)

```
MQCFST                       DSECT
MQCFST_TYPE                  DS   F        Structure type
MQCFST_STRUCLENGTH           DS   F        Structure length
MQCFST_PARAMETER             DS   F        Parameter identifier
MQCFST_CODEDCHARSETID        DS   F        Coded character set
*                                          identifier
MQCFST_STRINGLENGTH          DS   F        Length of string
MQCFST_LENGTH                EQU  *-MQCFST Length of structure
                             ORG  MQCFST
MQCFST_AREA                  DS   CL(MQCFST_LENGTH)
```

Visual Basic language declaration (MQCFST)

```
Type MQCFST
  Type As Long            ' Structure type
  StrucLength As Long     ' Structure length
  Parameter As Long       ' Parameter identifier
  CodedCharSetId As Long  ' Coded character set identifier
  StringLength As Long    ' Length of string
End Type

Global MQCFST_DEFAULT As MQCFST
```

Appendix B. Constants

This appendix specifies the values of the named constants that apply to events.

The constants are grouped according to the parameter or field to which they relate. All of the names of the constants in a group begin with a common prefix of the form "MQ*xxxx*_", where *xxxx* represents a string of 0 through 4 characters that indicates the nature of the values defined in that group. The constants are ordered alphabetically by the prefix.

Notes:

1. For constants with numeric values, the values are shown in both decimal and hexadecimal forms.
2. Hexadecimal values are represented using the notation X'hhhh', where each "h" denotes a single hexadecimal digit.
3. Character values are shown delimited by single quotation marks; the quotation marks are not part of the value.
4. Blanks in character values are represented by one or more occurrences of the symbol "b".

List of constants

The following sections list all of the named constants mentioned in this book, and show their values.

MQ_* (Lengths of character string and byte fields)

MQ_APPL_NAME_LENGTH	28	X'0000001C'
MQ_BRIDGE_NAME_LENGTH	24	X'00000018'
MQ_CHANNEL_NAME_LENGTH	20	X'00000014'
MQ_CONN_NAME_LENGTH	264	X'00000108'
MQ_FORMAT_LENGTH	8	X'00000008'
MQ_PROCESS_NAME_LENGTH	48	X'00000030'
MQ_Q_MGR_NAME_LENGTH	48	X'00000030'
MQ_Q_NAME_LENGTH	48	X'00000030'
MQ_USER_ID_LENGTH	12	X'0000000C'

MQBT_* (Bridge type)

MQBT_OTMA	1	X'00000001'

MQCA_* (Character attribute selector)

MQCA_BASE_Q_NAME	2002	X'000007D2'
MQCA_PROCESS_NAME	2012	X'000007DC'
MQCA_Q_MGR_NAME	2015	X'000007DF'
MQCA_Q_NAME	2016	X'000007E0'
MQCA_XMIT_Q_NAME	2024	X'000007E8'

MQCACF_* (Character attribute command format parameter)

MQCACF_OBJECT_Q_MGR_NAME	3023	X'00000BCF'
MQCACF_APPL_NAME	3024	X'00000BD0'
MQCACF_USER_IDENTIFIER	3025	X'00000BD1'
MQCACF_AUX_ERROR_DATA_STR_1	3026	X'00000BD2'
MQCACF_AUX_ERROR_DATA_STR_2	3027	X'00000BD3'
MQCACF_AUX_ERROR_DATA_STR_3	3028	X'00000BD4'
MQCACF_BRIDGE_NAME	3029	X'00000BD5'

MQCACH_* (Channel character attribute command format parameter)

MQCACH_CHANNEL_NAME	3501	X'00000DAD'
MQCACH_XMIT_Q_NAME	3505	X'00000DB1'
MQCACH_CONNECTION_NAME	3506	X'00000DB2'
MQCACH_FORMAT_NAME	3533	X'00000DCD'

MQCC_* (Completion code)

MQCC_OK	0	X'00000000'
MQCC_WARNING	1	X'00000001'

MQCFC_* (Command format control options)

MQCFC_LAST	1	X'00000001'

MQCFH_* (Command format header structure length)

MQCFH_STRUC_LENGTH	36	X'00000024'

MQCFH_* (Command format header version)

MQCFH_VERSION_1	1	X'00000001'

MQCFIN_* (Command format integer parameter structure length)

MQCFIN_STRUC_LENGTH	16	X'00000010'

MQCFST_* (Command format string parameter structure length)

MQCFST_STRUC_LENGTH_FIXED	20	X'00000014'

MQCFT_* (Command structure type)

MQCFT_COMMAND	1	X'00000001'
MQCFT_INTEGER	3	X'00000003'
MQCFT_STRING	4	X'00000004'
MQCFT_EVENT	7	X'00000007'

MQCHT_* (Channel type)

MQCHT_RECEIVER	3	X'00000003'
MQCHT_SVRCONN	7	X'00000007'
MQCHT_CLUSSDR	9	X'00000009'

MQCMD_* (Command identifier)

MQCMD_Q_MGR_EVENT	44	X'0000002C'
MQCMD_PERFM_EVENT	45	X'0000002D'
MQCMD_CHANNEL_EVENT	46	X'0000002E'

MQIA_* (Integer attribute selector)

MQIA_APPL_TYPE	1	X'00000001'
MQIA_Q_TYPE	20	X'00000014'
MQIA_TIME_SINCE_RESET	35	X'00000023'
MQIA_HIGH_Q_DEPTH	36	X'00000024'
MQIA_MSG_ENQ_COUNT	37	X'00000025'
MQIA_MSG_DEQ_COUNT	38	X'00000026'
MQIA_Q_DEPTH_HIGH_LIMIT	40	X'00000028'
MQIA_Q_DEPTH_LOW_LIMIT	41	X'00000029'
MQIA_Q_DEPTH_MAX_EVENT	42	X'0000002A'
MQIA_Q_DEPTH_HIGH_EVENT	43	X'0000002B'
MQIA_Q_DEPTH_LOW_EVENT	44	X'0000002C'

MQIACF_* (Integer attribute command format parameter)

MQIACF_ERROR_IDENTIFIER	1013	X'000003F5'
MQIACF_REASON_QUALIFIER	1020	X'000003FC'
MQIACF_COMMAND	1021	X'000003FD'
MQIACF_OPEN_OPTIONS	1022	X'000003FE'
MQIACF_AUX_ERROR_DATA_INT_1	1070	X'0000042E'
MQIACF_AUX_ERROR_DATA_INT_2	1071	X'0000042F'
MQIACF_CONV_REASON_CODE	1072	X'00000430'
MQIACF_BRIDGE_TYPE	1073	X'00000431'

MQIACH_* (Channel Integer attribute command format parameter)

MQIACH_CHANNEL_TYPE	1511	X'000005E7'

MQQT_* (Queue type)

MQQT_MODEL	2	X'00000002'
MQQT_ALIAS	3	X'00000003'
MQQT_REMOTE	6	X'00000006'

MQRC_* (Reason code in MQCFH)

MQRC_ALIAS_BASE_Q_TYPE_ERROR	2001	X'000007D1'
MQRC_BRIDGE_STARTED	2125	X'0000084D'
MQRC_BRIDGE_STOPPED	2126	X'0000084E'
MQRC_CHANNEL_ACTIVATED	2295	X'000008F7'
MQRC_CHANNEL_AUTO_DEF_ERROR	2234	X'000008BA'
MQRC_CHANNEL_AUTO_DEF_OK	2233	X'000008B9'
MQRC_CHANNEL_CONV_ERROR	2284	X'000008EC'
MQRC_CHANNEL_NOT_ACTIVATED	2296	X'000008F8'
MQRC_CHANNEL_STARTED	2282	X'000008EA'
MQRC_CHANNEL_STOPPED	2283	X'000008EB'
MQRC_CHANNEL_STOPPED_BY_USER	2279	X'000008E7'
MQRC_DEF_XMIT_Q_TYPE_ERROR	2198	X'00000896'
MQRC_DEF_XMIT_Q_USAGE_ERROR	2199	X'00000897'
MQRC_GET_INHIBITED	2016	X'000007E0'
MQRC_NOT_AUTHORIZED	2035	X'000007F3'
MQRC_PUT_INHIBITED	2051	X'00000803'
MQRC_Q_DEPTH_HIGH	2224	X'000008B0'
MQRC_Q_DEPTH_LOW	2225	X'000008B1'
MQRC_Q_FULL	2053	X'00000805'
MQRC_Q_MGR_ACTIVE	2222	X'000008AE'
MQRC_Q_MGR_NOT_ACTIVE	2223	X'000008AF'
MQRC_Q_SERVICE_INTERVAL_HIGH	2226	X'000008B2'
MQRC_Q_SERVICE_INTERVAL_OK	2227	X'000008B3'
MQRC_Q_TYPE_ERROR	2057	X'00000809'
MQRC_REMOTE_Q_NAME_ERROR	2184	X'00000888'
MQRC_XMIT_Q_TYPE_ERROR	2091	X'0000082B'
MQRC_XMIT_Q_USAGE_ERROR	2092	X'0000082C'
MQRC_UNKOWN_ALIAS_BASE_Q	2082	X'00000822'
MQRC_UNKNOWN_DEF_XMIT_Q	2197	X'00000895'
MQRC_UNKNOWN_OBJECT_NAME	2085	X'00000825'
MQRC_UNKNOWN_REMOTE_Q_MGR	2087	X'00000827'
MQRC_UNKNOWN_XMIT_Q	2196	X'00000894'

MQRCCF_* (Reason code for command format)

MQRCCF_SUPPRESSED_BY_EXIT	4085	X'00000FF5'

MQRQ_* (Reason qualifier)

MQRQ_CONN_NOT_AUTHORIZED	1	X'00000001'
MQRQ_OPEN_NOT_AUTHORIZED	2	X'00000002'
MQRQ_CLOSE_NOT_AUTHORIZED	3	X'00000003'
MQRQ_CMD_NOT_AUTHORIZED	4	X'00000004'
MQRQ_Q_MGR_STOPPING	5	X'00000005'
MQRQ_Q_MGR_QUIESCING	6	X'00000006'

MQRQ_CHANNEL_STOPPED_OK	7	X'00000007'
MQRQ_CHANNEL_STOPPED_ERROR	8	X'00000008'
MQRQ_CHANNEL_STOPPED_RETRY	9	X'00000009'
MQRQ_CHANNEL_STOPPED_DISABLED	10	X'0000000A'
MQRQ_BRIDGE_STOPPED_OK	11	X'0000000B'
MQRQ_BRIDGE_STOPPED_ERROR	12	X'0000000C'

Constants

Appendix C. Header, COPY, and INCLUDE files

Various header, COPY, and INCLUDE files are provided to assist applications with the processing of event messages. These are described below for each of the supported programming languages. Not all of the files are available in all environments.

See:
- "C header files"
- "COBOL COPY files"
- "PL/I INCLUDE files" on page 136
- "System/390 Assembler COPY files" on page 136
- "Visual Basic header files" on page 136

C header files

The following header files are provided for the C programming language.

Table 14. C header files

Filename	Contents relating to this book
CMQC	Elementary data types, some named constants for events
CMQCFC	Additional named constants for events
CMQXC	Named constants for events relating to channels

COBOL COPY files

The following COPY files are provided for the COBOL programming language. Two COPY files are provided for each structure; one COPY file has initial values, the other does not.

Table 15. COBOL COPY files

File name (with initial values)	File name (without initial values)	Contents relating to this book
CMQV	–	Some named constants for events (not available on DOS clients and Windows clients)
CMQCFV	–	Additional named constants for events (available only on OS/390 and OS/400)
CMQXV	–	Named constants for events relating to channels (available only on OS/390 and AS/400)
CMQCFHV	CMQCFHL	Header structure for events (available only on OS/390)
CMQCFINV	CMQCFINL	Single-integer parameter structure for events (available only on OS/390)
CMQCFSTV	CMQCFSTL	Single-string parameter structure for events (available only on OS/390)

PL/I INCLUDE files

The following INCLUDE files are provided for the PL/I programming language. These files are available only on AIX, OS/390, OS/2, and Windows NT.

Table 16. PL/I INCLUDE files

Filename	Contents relating to this book
CMQP	Some named constants for events
CMQCFP	Additional named constants for events
CMQXP	Named constants for events relating to channels

System/390 Assembler COPY files

The following COPY files are provided for the System/390 Assembler programming language. These files are available only on OS/390.

Table 17. System/390 Assembler COPY files

Filename	Contents relating to this book
CMQA	Some named constants for events
CMQCFA	Additional named constants for events
CMQXA	Named constants for events relating to channels
CMQCFHA	Header structure for events
CMQCFINA	Single-integer parameter structure for events
CMQCFSTA	Single-string parameter structure for events

Visual Basic header files

The following .BAS files are provided for the Visual Basic programming language. These files are available only on Windows platforms.

Table 18. Visual Basic header files

Filename	Contents relating to this book
CMQB	Some named constants for events
CMQCFB	Additional named constants for events
CMQXB	Named constants for events relating to channels

Appendix D. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Glossary of terms and abbreviations

This glossary defines MQSeries terms and abbreviations used in this book. If you do not find the term you are looking for, see the Index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with MQSeries for OS/390. A complete list of MQSeries for OS/390 abend reason codes and their explanations is contained in the *MQSeries for OS/390 Messages and Codes* manual.

active log. See *recovery log*.

adapter. An interface between MQSeries for OS/390 and TSO, IMS™, CICS, or batch address spaces. An adapter is an attachment facility that enables applications to access MQSeries services.

address space. The area of virtual storage available for a particular job.

address space identifier (ASID). A unique, system-assigned identifier for an address space.

administrator commands. MQSeries commands used to manage MQSeries objects, such as queues, processes, and namelists.

alert. A message sent to a management services focal point in a network to identify a problem or an impending problem.

alert monitor. In MQSeries for OS/390, a component of the CICS adapter that handles unscheduled events occurring as a result of connection requests to MQSeries for OS/390.

alias queue object. An MQSeries object, the name of which is an alias for a base queue defined to the local queue manager. When an application or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the associated base queue.

allied address space. See *ally*.

ally. An OS/390 address space that is connected to MQSeries for OS/390.

alternate user security. A security feature in which the authority of one user ID can be used by another user ID; for example, to open an MQSeries object.

APAR. Authorized program analysis report.

application environment. The software facilities that are accessible by an application program. On the OS/390 platform, CICS and IMS are examples of application environments.

application log. In Windows NT, a log that records significant application events.

application queue. A queue used by an application.

archive log. See *recovery log*.

ASID. Address space identifier.

asynchronous messaging. A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

attribute. One of a set of properties that defines the characteristics of an MQSeries object.

authorization checks. Security checks that are performed when a user tries to issue administration commands against an object, for example to open a queue or connect to a queue manager.

authorization file. In MQSeries on UNIX systems, a file that provides security definitions for an object, a class of objects, or all classes of objects.

authorization service. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a service that provides authority checking of commands and MQI calls for the user identifier associated with the command or call.

authorized program analysis report (APAR). A report of a problem caused by a suspected defect in a current, unaltered release of a program.

B

backout. An operation that reverses all the changes made during the current unit of recovery or unit of work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *commit*.

bag. See *data bag*.

basic mapping support (BMS). An interface between CICS and application programs that formats input and output display data and routes multiple-page output messages without regard for control characters used by various terminals.

BMS. Basic mapping support.

bootstrap data set (BSDS). A VSAM data set that contains:

- An inventory of all active and archived log data sets known to MQSeries for OS/390
- A wrap-around inventory of all recent MQSeries for OS/390 activity

The BSDS is required if the MQSeries for OS/390 subsystem has to be restarted.

browse. In message queuing, to use the MQGET call to copy a message without removing it from the queue. See also *get*.

browse cursor. In message queuing, an indicator used when browsing a queue to identify the message that is next in sequence.

BSDS. Bootstrap data set.

buffer pool. An area of main storage used for MQSeries for OS/390 queues, messages, and object definitions. See also *page set*.

C

call back. In MQSeries, a requester message channel initiates a transfer from a sender channel by first calling the sender, then closing down and awaiting a call back.

CCF. Channel control function.

CCSID. Coded character set identifier.

CDF. Channel definition file.

channel. See *message channel*.

channel control function (CCF). In MQSeries, a program to move messages from a transmission queue to a communication link, and from a communication link to a local queue, together with an operator panel interface to allow the setup and control of channels.

channel definition file (CDF). In MQSeries, a file containing communication channel definitions that associate transmission queues with communication links.

channel event. An event indicating that a channel instance has become available or unavailable. Channel events are generated on the queue managers at both ends of the channel.

checkpoint. A time when significant information is written on the log. Contrast with *syncpoint*. In MQSeries on UNIX systems, the point in time when a data record described in the log is the same as the data record in the queue. Checkpoints are generated automatically and are used during the system restart process.

CI. Control interval.

circular logging. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the process of keeping all restart data in a ring of log files. Logging fills the first file in the ring and then moves on to the next, until all the files are full. At this point, logging goes back to the first file in the ring and starts again, if the space has been freed or is no longer needed. Circular logging is used during restart recovery, using the log to roll back transactions that were in progress when the system stopped. Contrast with *linear logging*.

CL. Control Language.

client. A run-time component that provides access to queuing services on a server for local user applications. The queues used by the applications reside on the server. See also *MQSeries client*.

client application. An application, running on a workstation and linked to a client, that gives the application access to queuing services on a server.

client connection channel type. The type of MQI channel definition associated with an MQSeries client. See also *server connection channel type*.

cluster. A network of queue managers that are logically associated in some way.

coded character set identifier (CCSID). The name of a coded set of characters and their code point assignments.

command. In MQSeries, an administration instruction that can be carried out by the queue manager.

command prefix (CPF). In MQSeries for OS/390, a character string that identifies the queue manager to which MQSeries for OS/390 commands are directed, and from which MQSeries for OS/390 operator messages are received.

command processor. The MQSeries component that processes commands.

command server. The MQSeries component that reads commands from the system-command input queue, verifies them, and passes valid commands to the command processor.

commit. An operation that applies all the changes made during the current unit of recovery or unit of work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *backout*.

completion code. A return code indicating how an MQI call has ended.

configuration file. In MQSeries on UNIX systems, MQSeries for AS/400, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a file that contains configuration information related to, for example, logs, communications, or installable services. Synonymous with *.ini file*. See also *stanza*.

connect. To provide a queue manager connection handle, which an application uses on subsequent MQI calls. The connection is made either by the MQCONN or MQCONNX call, or automatically by the MQOPEN call.

connection handle. The identifier or token by which a program accesses the queue manager to which it is connected.

context. Information about the origin of a message.

context security. In MQSeries, a method of allowing security to be handled such that messages are obliged to carry details of their origins in the message descriptor.

control command. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a command that can be entered interactively from the operating system command line. Such a command requires only that the MQSeries product be installed; it does not require a special utility or program to run it.

control interval (CI). A fixed-length area of direct access storage in which VSAM stores records and creates distributed free spaces. The control interval is the unit of information that VSAM transmits to or from direct access storage.

Control Language (CL). In MQSeries for AS/400, a language that can be used to issue commands, either at the command line or by writing a CL program.

controlled shutdown. See *quiesced shutdown*.

CPF. Command prefix.

Cross Systems Coupling Facility (XCF). Provides the OS/390 coupling services that allow authorized programs in a multisystem environment to communicate with programs on the same or different OS/390 systems.

coupling facility. On OS/390, a special logical partition that provides high-speed caching, list processing, and locking functions in a parallel sysplex.

D

DAE. Dump analysis and elimination.

data bag. In the MQAI, a bag that allows you to handle properties (or parameters) of objects.

data item. In the MQAI, an item contained within a data bag. This can be an integer item or a character-string item, and a user item or a system item.

data conversion interface (DCI). The MQSeries interface to which customer- or vendor-written programs that convert application data between different machine encodings and CCSIDs must conform. A part of the MQSeries Framework.

datagram. The simplest message that MQSeries supports. This type of message does not require a reply.

DCE. Distributed Computing Environment.

DCI. Data conversion interface.

dead-letter queue (DLQ). A queue to which a queue manager or application sends messages that it cannot deliver to their correct destination.

dead-letter queue handler. An MQSeries-supplied utility that monitors a dead-letter queue (DLQ) and processes messages on the queue in accordance with a user-written rules table.

default object. A definition of an object (for example, a queue) with all attributes defined. If a user defines an object but does not specify all possible attributes for that object, the queue manager uses default attributes in place of any that were not specified.

deferred connection. A pending event that is activated when a CICS subsystem tries to connect to MQSeries for OS/390 before MQSeries for OS/390 has been started.

distributed application. In message queuing, a set of application programs that can each be connected to a different queue manager, but that collectively constitute a single application.

Distributed Computing Environment (DCE). Middleware that provides some basic services, making the development of distributed applications easier. DCE is defined by the Open Software Foundation (OSF).

distributed queue management (DQM). In message queuing, the setup and control of message channels to queue managers on other systems.

DLQ. Dead-letter queue.

DQM. Distributed queue management.

dual logging. A method of recording MQSeries for OS/390 activity, where each change is recorded on two data sets, so that if a restart is necessary and one data set is unreadable, the other can be used. Contrast with *single logging*.

dual mode. See *dual logging*.

dump analysis and elimination (DAE). An OS/390 service that enables an installation to suppress SVC dumps and ABEND SYSUDUMP dumps that are not needed because they duplicate previously written dumps.

dynamic queue. A local queue created when a program opens a model queue object. See also *permanent dynamic queue* and *temporary dynamic queue*.

E

environment. See *application environment*.

ESM. External security manager.

ESTAE. Extended specify task abnormal exit.

event. See *channel event*, *instrumentation event*, *performance event*, and *queue manager event*.

event data. In an event message, the part of the message data that contains information about the event (such as the queue manager name, and the application that gave rise to the event). See also *event header*.

event header. In an event message, the part of the message data that identifies the event type of the reason code for the event.

event log. See *application log*.

event message. Contains information (such as the category of event, the name of the application that caused the event, and queue manager statistics) relating to the origin of an instrumentation event in a network of MQSeries systems.

event queue. The queue onto which the queue manager puts an event message after it detects an event. Each category of event (queue manager, performance, or channel event) has its own event queue.

Event Viewer. A tool provided by Windows NT to examine and manage log files.

extended specify task abnormal exit (ESTAE). An OS/390 macro that provides recovery capability and gives control to the specified exit routine for processing, diagnosing an abend, or specifying a retry address.

external security manager (ESM). A security product that is invoked by the OS/390 System Authorization Facility. RACF® is an example of an ESM.

F

FFST™. First Failure Support Technology™.

FIFO. First-in-first-out.

First Failure Support Technology (FFST). Used by MQSeries on UNIX systems, MQSeries for OS/2 Warp, MQSeries for Windows NT, and MQSeries for AS/400 to detect and report software problems.

first-in-first-out (FIFO). A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

forced shutdown. A type of shutdown of the CICS adapter where the adapter immediately disconnects from MQSeries for OS/390, regardless of the state of any currently active tasks. Contrast with *quiesced shutdown*.

Framework. In MQSeries, a collection of programming interfaces that allow customers or vendors to write programs that extend or replace certain functions provided in MQSeries products. The interfaces are:

- MQSeries data conversion interface (DCI)
- MQSeries message channel interface (MCI)
- MQSeries name service interface (NSI)
- MQSeries security enabling interface (SEI)
- MQSeries trigger monitor interface (TMI)

FRR. Functional recovery routine.

functional recovery routine (FRR). An OS/390 recovery/termination manager facility that enables a recovery routine to gain control in the event of a program interrupt.

G

GCPC. Generalized command preprocessor.

generalized command preprocessor (GCPC). An MQSeries for OS/390 component that processes MQSeries commands and runs them.

Generalized Trace Facility (GTF). An OS/390 service program that records significant system events, such as supervisor calls and start I/O operations, for the purpose of problem determination.

get. In message queuing, to use the MQGET call to remove a message from a queue.

global trace. An MQSeries for OS/390 trace option where the trace data comes from the entire MQSeries for OS/390 subsystem.

globally-defined object. On OS/390, an object whose definition is stored in the shared repository. The object is available to all queue managers in the queue-sharing group. See also *locally-defined object*.

GTF. Generalized Trace Facility.

H

handle. See *connection handle* and *object handle*.

hardened message. A message that is written to auxiliary (disk) storage so that the message will not be lost in the event of a system failure. See also *persistent message*.

I

ILE. Integrated Language Environment®.

immediate shutdown. In MQSeries, a shutdown of a queue manager that does not wait for applications to disconnect. Current MQI calls are allowed to complete, but new MQI calls fail after an immediate shutdown has been requested. Contrast with *quiesced shutdown* and *preemptive shutdown*.

inbound channel. A channel that receives messages from another queue manager. See also *shared inbound channel*.

in-doubt unit of recovery. In MQSeries, the status of a unit of recovery for which a syncpoint has been requested but not yet confirmed.

Integrated Language Environment® (ILE). The AS/400 Integrated Language Environment. This replaces the AS/400 Original Program Model (OPM).

.ini file. See *configuration file*.

initialization input data sets. Data sets used by MQSeries for OS/390 when it starts up.

initiation queue. A local queue on which the queue manager puts trigger messages.

input/output parameter. A parameter of an MQI call in which you supply information when you make the call, and in which the queue manager changes the information when the call completes or fails.

input parameter. A parameter of an MQI call in which you supply information when you make the call.

installable services. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, additional functionality provided as independent components. The installation of each component is optional: in-house or third-party components can be used instead. See also *authorization service*, *name service*, and *user identifier service*.

instrumentation event. A facility that can be used to monitor the operation of queue managers in a network of MQSeries systems. MQSeries provides instrumentation events for monitoring queue manager resource definitions, performance conditions, and channel conditions. Instrumentation events can be used by a user-written reporting mechanism in an administration application that displays the events to a system operator. They also allow applications acting as agents for other administration networks to monitor reports and create the appropriate alerts.

Interactive Problem Control System (IPCS). A component of OS/390 that permits online problem management, interactive problem diagnosis, online debugging for disk-resident abend dumps, problem tracking, and problem reporting.

Interactive System Productivity Facility (ISPF). An IBM licensed program that serves as a full-screen editor and dialog manager. It is used for writing application programs, and provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user.

IPCS. Interactive Problem Control System.

ISPF. Interactive System Productivity Facility.

L

linear logging. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the process of keeping restart data in a sequence of files. New files are added to the sequence as necessary. The space in which the data is written is not reused until the queue manager is restarted. Contrast with *circular logging*.

listener. In MQSeries distributed queuing, a program that monitors for incoming network connections.

local definition. An MQSeries object belonging to a local queue manager.

local definition of a remote queue. An MQSeries object belonging to a local queue manager. This object defines the attributes of a queue that is owned by another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

local queue. A queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *remote queue*.

local queue manager. The queue manager to which a program is connected and that provides message queuing services to the program. Queue managers to which a program is not connected are called *remote queue managers*, even if they are running on the same system as the program.

locale. On UNIX systems, a subset of a user's environment that defines conventions for a specific culture (such as time, numeric, or monetary formatting and character classification, collation, or conversion). The queue manager CCSID is derived from the locale of the user ID that created the queue manager.

locally-defined object. On OS/390, an object whose definition is stored on page set zero. The definition can be accessed only by the queue manager that defined it. Also known as a *privately-defined object*.

log. In MQSeries, a file recording the work done by queue managers while they receive, transmit, and deliver messages, to enable them to recover in the event of failure.

log control file. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the file containing information needed to monitor the use of log files (for example, their size and location, and the name of the next available file).

log file. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a file in which all significant changes to the data controlled by a queue manager are recorded. If the primary log files become full, MQSeries allocates secondary log files.

logical unit of work (LUW). See *unit of work*.

M

machine check interrupt. An interruption that occurs as a result of an equipment malfunction or error. A machine check interrupt can be either hardware recoverable, software recoverable, or nonrecoverable.

MCA. Message channel agent.

MCI. Message channel interface.

media image. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the sequence of log records that contain an image of an object. The object can be recreated from this image.

message. In message queuing applications, a communication sent between programs. In system programming, information intended for the terminal operator or system administrator.

message channel. In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender at one end and a receiver at the other end) and a communication link. Contrast with *MQI channel*.

message channel agent (MCA). A program that transmits prepared messages from a transmission queue to a communication link, or from a communication link to a destination queue. See also *message queue interface*.

message channel interface (MCI). The MQSeries interface to which customer- or vendor-written programs that transmit messages between an MQSeries queue manager and another messaging system must conform. A part of the MQSeries Framework.

message descriptor. Control information describing the message format and presentation that is carried as part of an MQSeries message. The format of the message descriptor is defined by the MQMD structure.

message priority. In MQSeries, an attribute of a message that can affect the order in which messages on a queue are retrieved, and whether a trigger event is generated.

message queue. Synonym for *queue*.

message queue interface (MQI). The programming interface provided by the MQSeries queue managers. This programming interface allows application programs to access message queuing services.

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

message sequence numbering. A programming technique in which messages are given unique numbers during transmission over a communication link. This enables the receiving process to check whether all messages are received, to place them in a queue in the original order, and to discard duplicate messages.

messaging. See *synchronous messaging* and *asynchronous messaging*.

model queue object. A set of queue attributes that act as a template when a program creates a dynamic queue.

MQAI. MQSeries Administration Interface.

MQI. Message queue interface.

MQI channel. Connects an MQSeries client to a queue manager on a server system, and transfers only MQI calls and responses in a bidirectional manner. Contrast with *message channel*.

MQSC. MQSeries commands.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries Administration Interface (MQAI). A programming interface to MQSeries.

MQSeries client. Part of an MQSeries product that can be installed on a system without installing the full queue manager. The MQSeries client accepts MQI calls from applications and communicates with a queue manager on a server system.

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects.

N

namelist. An MQSeries object that contains a list of names, for example, queue names.

name service. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, the facility that determines which queue manager owns a specified queue.

name service interface (NSI). The MQSeries interface to which customer- or vendor-written programs that resolve queue-name ownership must conform. A part of the MQSeries Framework.

name transformation. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, an internal process that changes a queue manager name so that it is unique and valid for the system being used. Externally, the queue manager name remains unchanged.

New Technology File System (NTFS). A Windows NT recoverable file system that provides security for files.

nonpersistent message. A message that does not survive a restart of the queue manager. Contrast with *persistent message*.

NSI. Name service interface.

NTFS. New Technology File System.

null character. The character that is represented by X'00'.

O

OAM. Object authority manager.

object. In MQSeries, an object is a queue manager, a queue, a process definition, a channel, a namelist, or a storage class (OS/390 only).

object authority manager (OAM). In MQSeries on UNIX systems, MQSeries for AS/400, and MQSeries for Windows NT, the default authorization service for command and object management. The OAM can be replaced by, or run in combination with, a customer-supplied security service.

object descriptor. A data structure that identifies a particular MQSeries object. Included in the descriptor are the name of the object and the object type.

object handle. The identifier or token by which a program accesses the MQSeries object with which it is working.

off-loading. In MQSeries for OS/390, an automatic process whereby a queue manager's active log is transferred to its archive log.

Open Transaction Manager Access (OTMA). A transaction-based, connectionless client/server protocol. It functions as an interface for host-based communications servers accessing IMS TM applications through the OS/390 Cross Systems Coupling Facility (XCF). OTMA is implemented in an OS/390 sysplex environment. Therefore, the domain of OTMA is restricted to the domain of XCF.

OPM. Original Program Model.

Original Program Model (OPM). The AS/400 Original Program Model. This is no longer supported on MQSeries. It is replaced by the Integrated Language Environment (ILE).

OTMA. Open Transaction Manager Access.

outbound channel. A channel that takes messages from a transmission queue and sends them to another queue manager. See also *shared outbound channel*.

output log-buffer. In MQSeries for OS/390, a buffer that holds recovery log records before they are written to the archive log.

output parameter. A parameter of an MQI call in which the queue manager returns information when the call completes or fails.

P

page set. A VSAM data set used when MQSeries for OS/390 moves data (for example, queues and messages) from buffers in main storage to permanent backing storage (DASD).

PCF. Programmable command format.

PCF command. See *programmable command format*.

pending event. An unscheduled event that occurs as a result of a connect request from a CICS adapter.

percolation. In error recovery, the passing along a preestablished path of control from a recovery routine to a higher-level recovery routine.

performance event. A category of event indicating that a limit condition has occurred.

performance trace. An MQSeries trace option where the trace data is to be used for performance analysis and tuning.

permanent dynamic queue. A dynamic queue that is deleted when it is closed only if deletion is explicitly requested. Permanent dynamic queues are recovered if the queue manager fails, so they can contain persistent messages. Contrast with *temporary dynamic queue*.

persistent message. A message that survives a restart of the queue manager. Contrast with *nonpersistent message*.

ping. In distributed queuing, a diagnostic aid that uses the exchange of a test message to confirm that a message channel or a TCP/IP connection is functioning.

platform. In MQSeries, the operating system under which a queue manager is running.

point of recovery. In MQSeries for OS/390, the term used to describe a set of backup copies of MQSeries for OS/390 page sets and the corresponding log data sets required to recover these page sets. These backup copies provide a potential restart point in the event of page set loss (for example, page set I/O error).

preemptive shutdown. In MQSeries, a shutdown of a queue manager that does not wait for connected applications to disconnect, nor for current MQI calls to complete. Contrast with *immediate shutdown* and *quiesced shutdown*.

principal. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a term used for a user identifier. Used by the object authority manager for checking authorizations to system resources.

privately-defined object. In OS/390, an object whose definition is stored on page set zero. The definition can be accessed only by the queue manager that defined it. Also known as a *locally-defined object*.

process definition object. An MQSeries object that contains the definition of an MQSeries application. For example, a queue manager uses the definition when it works with trigger messages.

programmable command format (PCF). A type of MQSeries message used by:

- User administration applications, to put PCF commands onto the system command input queue of a specified queue manager
- User administration applications, to get the results of a PCF command from a specified queue manager
- A queue manager, as a notification that an event has occurred

Contrast with *MQSC*.

program temporary fix (PTF). A solution or by-pass of a problem diagnosed by IBM field engineering as the result of a defect in a current, unaltered release of a program.

PTF. Program temporary fix.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages—they point to other queues, or can be used as models for dynamic queues.

queue manager. A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. An MQSeries object that defines the attributes of a particular queue manager.

queue manager event. An event that indicates:

- An error condition has occurred in relation to the resources used by a queue manager. For example, a queue is unavailable.
- A significant change has occurred in the queue manager. For example, a queue manager has stopped or started.

queue-sharing group. In MQSeries for OS/390, a group of queue managers in the same sysplex that can access a single set of object definitions stored in the shared repository, and a single set of shared queues stored in the coupling facility. See also *shared queue*.

queuing. See *message queuing*.

quiesced shutdown. In MQSeries, a shutdown of a queue manager that allows all connected applications to disconnect. Contrast with *immediate shutdown* and *preemptive shutdown*. A type of shutdown of the CICS adapter where the adapter disconnects from MQSeries, but only after all the currently active tasks have been completed. Contrast with *forced shutdown*.

quiescing. In MQSeries, the state of a queue manager prior to it being stopped. In this state, programs are allowed to finish processing, but no new programs are allowed to start.

R

RBA. Relative byte address.

reason code. A return code that describes the reason for the failure or partial success of an MQI call.

receiver channel. In message queuing, a channel that responds to a sender channel, takes messages from a communication link, and puts them on a local queue.

recovery log. In MQSeries for OS/390, data sets containing information needed to recover messages, queues, and the MQSeries subsystem. MQSeries for OS/390 writes each record to a data set called the *active log*. When the active log is full, its contents are off-loaded to a DASD or tape data set called the *archive log*. Synonymous with *log*.

recovery termination manager (RTM). A program that handles all normal and abnormal termination of tasks by passing control to a recovery routine associated with the terminating function.

Registry. In Windows NT, a secure database that provides a single source for system and application configuration data.

Registry Editor. In Windows NT, the program item that allows the user to edit the Registry.

Registry Hive. In Windows NT, the structure of the data stored in the Registry.

relative byte address (RBA). The displacement in bytes of a stored record or control interval from the beginning of the storage space allocated to the data set to which it belongs.

remote queue. A queue belonging to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager. To a program, a queue manager that is not the one to which the program is connected.

remote queue object. See *local definition of a remote queue*.

remote queuing. In message queuing, the provision of services to enable applications to put messages on queues belonging to other queue managers.

reply message. A type of message used for replies to request messages. Contrast with *request message* and *report message*.

reply-to queue. The name of a queue to which the program that issued an MQPUT call wants a reply message or report message sent.

report message. A type of message that gives information about another message. A report message can indicate that a message has been delivered, has arrived at its destination, has expired, or could not be processed for some reason. Contrast with *reply message* and *request message*.

requester channel. In message queuing, a channel that may be started remotely by a sender channel. The requester channel accepts messages from the sender channel over a communication link and puts the messages on the local queue designated in the message. See also *server channel*.

request message. A type of message used to request a reply from another program. Contrast with *reply message* and *report message*.

RESLEVEL. In MQSeries for OS/390, an option that controls the number of CICS user IDs checked for API-resource security in MQSeries for OS/390.

resolution path. The set of queues that are opened when an application specifies an alias or a remote queue on input to an MQOPEN call.

resource. Any facility of the computing system or operating system required by a job or task. In MQSeries for OS/390, examples of resources are buffer pools, page sets, log data sets, queues, and messages.

resource manager. An application, program, or transaction that manages and controls access to shared resources such as memory buffers and data sets. MQSeries, CICS, and IMS are resource managers.

responder. In distributed queuing, a program that replies to network connection requests from another system.

resynch. In MQSeries, an option to direct a channel to start up and resolve any in-doubt status messages, but without restarting message transfer.

return codes. The collective name for completion codes and reason codes.

rollback. Synonym for *back out*.

RTM. Recovery termination manager.

rules table. A control file containing one or more rules that the dead-letter queue handler applies to messages on the DLQ.

S

SAF. System Authorization Facility.

SDWA. System diagnostic work area.

security enabling interface (SEI). The MQSeries interface to which customer- or vendor-written programs that check authorization, supply a user identifier, or perform authentication must conform. A part of the MQSeries Framework.

SEI. Security enabling interface.

sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

sequential delivery. In MQSeries, a method of transmitting messages with a sequence number so that the receiving channel can reestablish the message sequence when storing the messages. This is required where messages must be delivered only once, and in the correct order.

sequential number wrap value. In MQSeries, a method of ensuring that both ends of a communication link reset their current message sequence numbers at the same time. Transmitting messages with a sequence number ensures that the receiving channel can reestablish the message sequence when storing the messages.

server. (1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *client*.

server channel. In message queuing, a channel that responds to a requester channel, removes messages from a transmission queue, and moves them over a communication link to the requester channel.

server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *client connection channel type*.

service interval. A time interval, against which the elapsed time between a put or a get and a subsequent get is compared by the queue manager in deciding whether the conditions for a service interval event have been met. The service interval for a queue is specified by a queue attribute.

service interval event. An event related to the service interval.

session ID. In MQSeries for OS/390, the CICS-unique identifier that defines the communication link to be used by a message channel agent when moving messages from a transmission queue to a link.

shared inbound channel. In MQSeries for OS/390, a channel that was started by a listener using the group port. The channel definition of a shared channel can be stored either on page set zero (private) or in the shared repository (global).

shared outbound channel. In MQSeries for OS/390, a channel that moves messages from a shared transmission queue. The channel definition of a shared channel can be stored either on page set zero (private) or in the shared repository (global).

shared queue. In MQSeries for OS/390, a type of local queue. The messages on the queue are stored in the *coupling facility* and can be accessed by one or more queue managers in a *queue-sharing group*. The definition of the queue is stored in the *shared repository*.

shared repository. In MQSeries for OS/390, a shared DB2® database that is used to hold object definitions that have been defined globally.

shutdown. See *immediate shutdown*, *preemptive shutdown*, and *quiesced shutdown*.

signaling. In MQSeries for OS/390 and MQSeries for Windows 2.1, a feature that allows the operating system to notify a program when an expected message arrives on a queue.

single logging. A method of recording MQSeries for OS/390 activity where each change is recorded on one data set only. Contrast with *dual logging*.

single-phase backout. A method in which an action in progress must not be allowed to finish, and all changes that are part of that action must be undone.

single-phase commit. A method in which a program can commit updates to a queue without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with *two-phase commit*.

SIT. System initialization table.

stanza. A group of lines in a configuration file that assigns a value to a parameter modifying the behavior of a queue manager, client, or channel. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a configuration (.ini) file may contain a number of stanzas.

storage class. In MQSeries for OS/390, a storage class defines the page set that is to hold the messages for a particular queue. The storage class is specified when the queue is defined.

store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

subsystem. In OS/390, a group of modules that provides function that is dependent on OS/390. For example, MQSeries for OS/390 is an OS/390 subsystem.

supervisor call (SVC). An OS/390 instruction that interrupts a running program and passes control to the supervisor so that it can perform the specific service indicated by the instruction.

SVC. Supervisor call.

switch profile. In MQSeries for OS/390, a RACF profile used when MQSeries starts up or when a refresh security command is issued. Each switch profile that MQSeries detects turns off checking for the specified resource.

symptom string. Diagnostic information displayed in a structured format designed for searching the IBM software support database.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

syncpoint. An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

System Authorization Facility (SAF). An OS/390 facility through which MQSeries for OS/390 communicates with an external security manager such as RACF.

system.command.input queue. A local queue on which application programs can put MQSeries commands. The commands are retrieved from the queue by the command server, which validates them and passes them to the command processor to be run.

system control commands. Commands used to manipulate platform-specific entities such as buffer pools, storage classes, and page sets.

system diagnostic work area (SDWA). Data recorded in a SYS1.LOGREC entry, which describes a program or hardware error.

system initialization table (SIT). A table containing parameters used by CICS on start up.

SYS1.LOGREC. A service aid containing information about program and hardware errors.

T

target library high-level qualifier (thlqual). High-level qualifier for OS/390 target data set names.

task control block (TCB). An OS/390 control block used to communicate information about tasks within an address space that are connected to an OS/390 subsystem such as MQSeries for OS/390 or CICS.

task switching. The overlapping of I/O operations and processing between several tasks. In MQSeries for OS/390, the task switcher optimizes performance by allowing some MQI calls to be executed under subtasks rather than under the main CICS TCB.

TCB. Task control block.

temporary dynamic queue. A dynamic queue that is deleted when it is closed. Temporary dynamic queues are not recovered if the queue manager fails, so they can contain nonpersistent messages only. Contrast with *permanent dynamic queue*.

termination notification. A pending event that is activated when a CICS subsystem successfully connects to MQSeries for OS/390.

thlqual. Target library high-level qualifier.

thread. In MQSeries, the lowest level of parallel execution available on an operating system platform.

time-independent messaging. See *asynchronous messaging*.

TMI. Trigger monitor interface.

trace. In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF).

tranid. See *transaction identifier*.

transaction identifier. In CICS, a name that is specified when the transaction is defined, and that is used to invoke the transaction.

transmission program. See *message channel agent*.

transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message. A message containing information about the program that a trigger monitor is to start.

trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

trigger monitor interface (TMI). The MQSeries interface to which customer- or vendor-written trigger monitor programs must conform. A part of the MQSeries Framework.

two-phase commit. A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with *single-phase commit*.

U

UIS. User identifier service.

undelivered-message queue. See *dead-letter queue*.

undo/redo record. A log record used in recovery. The redo part of the record describes a change to be made to an MQSeries object. The undo part describes how to back out the change if the work is not committed.

unit of recovery. A recoverable sequence of operations within a single resource manager. Contrast with *unit of work*.

unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Contrast with *unit of recovery*.

user identifier service (UIS). In MQSeries for OS/2 Warp, the facility that allows MQI applications to associate a user ID, other than the default user ID, with MQSeries messages.

utility. In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

X

XCF. Cross Systems Coupling Facility.

Bibliography

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries "family" books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX, V5.1
- MQSeries for AS/400, V5.1
- MQSeries for AT&T GIS UNIX, V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for SINIX and DC/OSx, V2.2
- MQSeries for Sun Solaris, V5.1
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Tandem NonStop Kernel, V2.2.0.1
- MQSeries for VSE/ESA, V2.1
- MQSeries for Windows, V2.0
- MQSeries for Windows, V2.1
- MQSeries for Windows NT, V5.1

The MQSeries cross-platform publications are:

- *MQSeries Brochure*, G511-1908
- *An Introduction to Messaging and Queuing*, GC33-0805
- *MQSeries Intercommunication*, SC33-1872
- *MQSeries Queue Manager Clusters*, SC34-5349
- *MQSeries Clients*, GC33-1632
- *MQSeries System Administration*, SC33-1873
- *MQSeries MQSC Command Reference*, SC33-1369
- *MQSeries Event Monitoring*, SC34-5760
- *MQSeries Programmable System Management*, SC33-1482
- *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390
- *MQSeries Messages*, GC33-1876
- *MQSeries Application Programming Guide*, SC33-0807
- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Programming Interfaces Reference Summary*, SX33-6095
- *MQSeries Using C++*, SC33-1877
- *MQSeries Using Java*™, SC34-5456
- *MQSeries Application Messaging Interface*, SC34-5604

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX, V5.1

MQSeries for AIX Quick Beginnings, GC33-1867

MQSeries for AS/400, V5.1

MQSeries for AS/400 Quick Beginnings, GC34-5557

MQSeries for AS/400 System Administration, SC34-5558

MQSeries for AS/400 Application Programming Reference (ILE RPG), SC34-5559

MQSeries for AT&T GIS UNIX, V2.2

MQSeries for AT&T GIS UNIX System Management Guide, SC33-1642

MQSeries for Compaq (DIGITAL) OpenVMS, V2.2.1.1

MQSeries for Digital OpenVMS System Management Guide, GC33-1791

MQSeries for Compaq Tru64 UNIX, V5.1

MQSeries for Compaq Tru64 UNIX Quick Beginnings, GC34-5684

MQSeries for HP-UX, V5.1

MQSeries for HP-UX Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp, V5.1

MQSeries for OS/2 Warp Quick Beginnings, GC33-1868

MQSeries for OS/390, V5.2

MQSeries for OS/390 Concepts and Planning Guide, GC34-5650

MQSeries for OS/390 System Setup Guide, SC34-5651

MQSeries for OS/390 System Administration Guide, SC34-5652

MQSeries for OS/390 Problem Determination Guide, GC34-5892

MQSeries for OS/390 Messages and Codes, GC34-5891

MQSeries for OS/390 Licensed Program Specifications, GC34-5893

MQSeries for OS/390 Program Directory

MQSeries link for R/3, Version 1.2

MQSeries link for R/3 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx, V2.2

MQSeries for SINIX and DC/OSx System Management Guide, GC33-1768

MQSeries for Sun Solaris, V5.1

MQSeries for Sun Solaris Quick Beginnings, GC33-1870

MQSeries for Sun Solaris, Intel Platform Edition, V5.1

MQSeries for Sun Solaris, Intel Platform Edition Quick Beginnings, GC34-5851

MQSeries for Tandem NonStop Kernel, V2.2.0.1

MQSeries for Tandem NonStop Kernel System Management Guide, GC33-1893

MQSeries for VSE/ESA, V2.1

MQSeries for VSE/ESA, Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA™ System Management Guide, GC34-5364

MQSeries for Windows, V2.0

MQSeries for Windows User's Guide, GC33-1822

MQSeries for Windows, V2.1

MQSeries for Windows User's Guide, GC33-1965

MQSeries for Windows NT, V5.1

MQSeries for Windows NT Quick Beginnings, GC34-5389

MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387

MQSeries LotusScript Extension, SC34-5404

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:
- MQSeries for AIX, V5.1
- MQSeries for AS/400, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.1
- MQSeries for Windows NT, V5.1 (compiled HTML)
- MQSeries link for R/3, V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

```
http://www.ibm.com/software/mqseries/
```

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

PDF versions of relevant MQSeries books are supplied with these MQSeries products:
- MQSeries for AIX, V5.1
- MQSeries for AS/400, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.1
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.1
- MQSeries for Windows NT, V5.1
- MQSeries link for R/3, V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

```
http://www.ibm.com/software/mqseries/
```

BookManager® format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows, Version 2.0 and MQSeries for Windows, Version 2.1.

MQSeries information available on the Internet

The MQSeries product family Web site is at:

```
http://www.ibm.com/software/mqseries/
```

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

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

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–870229
  - From within the U.K., use 01962–870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5760-01