MQSeries® for Compaq OpenVMS Alpha®

System Administration Guide

Version 5 Release 1

SC34-5884-00

MQSeries® for Compaq OpenVMS Alpha®

System Administration Guide

Version 5 Release 1

SC34-5884-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix L. Notices" on page 361.

First edition (May 2001)

This edition applies to MQSeries for Compaq OpenVMS Alpha, Version 5.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this book

MQSeries for Compaq OpenVMS Alpha, V5.1 –also referred to in this book as MQSeries for Compaq OpenVMS or MQSeries, as the context permits–is part of the MQSeries family of products. These products provide application programming services that enable application programs to communicate with each other using *message queues*. This form of communication is referred to as *commercial messaging*. The applications involved can exist on different nodes on a wide variety of machine and operating system types. They use a common application programming interface, called the Message Queuing Interface or MQI, so that programs developed on one platform can readily be transferred to another.

This book describes the system administration aspects of MQSeries for Compaq OpenVMS Alpha, V5.1 and the services it provides to support commercial messaging in an OpenVMS environment. This includes managing the queues that applications use to receive their messages, and ensuring that applications have access to the queues that they require.

Who this book is for

Primarily, this book is for system administrators, and system programmers who manage the configuration and administration tasks for MQSeries. It is also useful to application programmers who must have some understanding of MQSeries administration tasks.

What you need to know to understand this book

To use this book, you should have a good understanding of the OpenVMS operating system and associated utilities. You do not need to have worked with message queuing products before, but you should have an understanding of the basic concepts of message queuing.

How to use this book

The body of this book:

- Introduces MQSeries
- Describes day-to-day management of an OpenVMS system, addressing topics such as administration of local and remote MQSeries objects, security, transactional support, and problem determination.

Using the appendixes

The appendixes provide reference material. Some include information that will be incorporated in other MQSeries books at the next opportunity.

Information about MQSeries on the Internet

MQSeries URL

The URL of the MQSeries product family home page is:

```
http://www.ibm.com/software/ts/mqseries/
```

MQSeries on the Internet

What's new in MQSeries for Compaq OpenVMS Alpha, V5.1

This following new function is described in this edition of the *MQSeries for Compaq OpenVMS Alpha, V5.1 System Administration Guide*.

MQSeries queue manager clusters

MQSeries queue managers can be connected to form a cluster of queue managers. Within a cluster, queue managers can make the queues they host available to every other queue manager. Any queue manager can send a message to any other queue manager in the same cluster without the need for explicit channel definitions, remote queue definitions, or transmission queues for each destination. The main benefits of MQSeries clusters are:
- Fewer system administration tasks
- Increased availability
- Workload balancing

See "Clusters" on page 12 for more information.

Note: MQSeries clusters are not the same as OpenVMS clusters. When the term *cluster* is used, it refers to an MQSeries queue manager cluster. An OpenVMS cluster is always referred to as an *OpenVMS cluster*. For more on OpenVMS clusters, see "Chapter 16. MQSeries for OpenVMS and clustering" on page 203.

MQSeries Application Interface (MQAI)

MQSeries for Compaq OpenVMS now supports the MQSeries Application Interface (MQAI), a programming interface that simplifies the use of PCF messages to configure MQSeries. For more information about the MQAI, including full command descriptions, see *MQSeries Administration Interface Programming Guide and Reference*.

Message queue size

A message queue can be up to 2 GB.

Controlled, wait shutdown of a queue manager

A new option has been added to the **endmqm** command to allow controlled, synchronous shutdown of a queue manager.

Java® support

MQSeries for Compaq OpenVMS now works with Java compilers.

Web administration

You can now perform remotely the following administration tasks for MQSeries for Compaq OpenVMS using an HTML browser (for example, Netscape Navigator or Microsoft® Internet Explorer) on a Windows NT ® system:
- Log on as an MQSeries Administrator
- Select a queue manager and issue MQSC commands against it
- Create, edit and delete MQSC scripts.

Part 1. Guidance

Chapter 1. Introduction to MQSeries

This chapter introduces MQSeries for Compaq OpenVMS from an administrator's perspective, and describes the basic concepts of MQSeries and messaging. It contains these sections:

MQSeries and message queuing

MQSeries allows application programs to use message queuing to participate in message-driven processing. Application programs can communicate across different platforms by using the appropriate message queuing software products. For example, OpenVMS and MVS/ESA® applications can communicate through MQSeries for Compaq OpenVMS and MQSeries for OS/390® respectively. The applications are shielded from the mechanics of the underlying communications.

MQSeries products implement a common application programming interface known as the message queue interface (MQI) whatever platform the applications are run on. This makes it easier for you to port applications from one platform to another.

The MQI is described in detail in the *MQSeries Application Programming Reference* book.

Time-independent applications

With message queuing, the exchange of messages between the sending and receiving programs is independent of time. This means that the sending and receiving applications are decoupled so that the sender can continue processing without having to wait for the receiver to acknowledge the receipt of the message. In fact, the target application does not even have to be running when the message is sent. It can retrieve the message after it has been started.

Message-driven processing

Upon arrival on a queue, messages can automatically start an application using a mechanism known as *triggering*. If necessary, the applications can be stopped when the message or messages have been processed.

Messages and queues

Messages and queues are the basic components of a message queuing system.

What is a message?

A *message* is a string of bytes that is meaningful to the applications that use it. Messages are used for transferring information from one application to another (or to different parts of the same application). The applications can be running on the same platform, or on different platforms.

MQSeries messages have two parts:

- *application data*

  The content and structure of the application data is defined by the application programs that use them.
- *message descriptor*

  The message descriptor identifies the message and contains other control information, such as the type of message and the priority assigned to the message by the sending application.

  The format of the message descriptor is defined by MQSeries. For a complete description of the message descriptor, see the *MQSeries Application Programming Reference* guide.

Message lengths

In MQSeries, the maximum message length is 100 MB (where 1 MB equals 1 048 576 bytes). In practice, the message length may be limited by:

- The maximum message length defined for the receiving queue.
- The maximum message length defined for the queue manager.
- The maximum message length defined by either the sending or receiving application.
- The amount of storage available for the message.

It may take several messages to send all the information that an application requires.

What is a queue?

A *queue* is a data structure used to store messages. The messages may be put on the queue by application programs or by a queue manager as part of its normal operation.

Each queue is owned by a *queue manager*. The queue manager is responsible for maintaining the queues it owns and for storing all the messages it receives onto the appropriate queues.

The maximum size of a queue is 2 GB. For information about planning the amount of storage you require for queues, see the *MQSeries Planning Guide* or visit the following web site for platform-specific performance reports:

http://www.ibm.com/software/ts/mqseries/txppacs/txpm1.html

How do applications send and receive messages?

Applications send and receive messages using *MQI calls*. For example, to put a message onto a queue, an application:

1. Opens the required queue by issuing an MQI MQOPEN call.
2. Issues an MQI MQPUT call to put the message onto the queue
3. Another application can retrieve the message from the same queue by issuing an MQI MQGET call.

For more information about MQI calls, see the *MQSeries Application Programming Reference* book.

Predefined and dynamic queues

Queues can be characterized by the way they are created:

- *Predefined queues* are created by an administrator using the appropriate command set. For example, the MQSC command DEFINE QLOCAL creates a predefined local queue. Predefined queues are permanent; they exist independently of the applications that use them and survive MQSeries restarts.
- *Dynamic queues* are created when an application issues an OPEN request specifying the name of a *model queue*. The queue created is based on a template queue definition, which is the model queue. You can create a model queue using the MQSC command DEFINE QMODEL. The attributes of a model queue, for example the maximum number of messages that can be stored on it, are inherited by any dynamic queue that is created from it.

  Model queues have an attribute that specifies whether the dynamic queue is to be permanent or temporary. Permanent queues survive application and queue manager restarts; temporary queues are lost on a restart.

Retrieving messages from queues

In MQSeries, suitably authorized applications can retrieve messages from a queue according to these retrieval algorithms:

- First-in-first-out (FIFO).
- Message priority, as defined in the message descriptor. Messages that have the same priority are retrieved on a FIFO basis.
- A program request for a specific message.

The MQGET request from the application determines the method used.

Objects

Many of the tasks described in this book involve manipulating MQSeries *objects*. In MQSeries Version 5.1, the object types include queue managers, queues, process definitions, channels, clusters, and namelists.

The manipulation or *administration* of objects includes:

- Starting and stopping queue managers
- Creating objects, particularly queues, for applications.
- Working with channels to create communication paths to queue managers on other (remote) systems. This is described in detail in the *MQSeries Intercommunication* book.
- Creating *clusters* of queue managers to simplify the overall administration process, or to achieve workload balancing.

This book contains detailed information about administration in the following chapters:

- “Chapter 2. An introduction to MQSeries administration” on page 17
- “Chapter 3. Managing queue managers using control commands” on page 23
- “Chapter 4. Administering local MQSeries objects” on page 31
- “Chapter 6. Administering remote MQSeries objects” on page 59

Object names

The naming convention adopted for MQSeries objects depends on the object.

Each instance of a queue manager is known by its name. This name must be unique within the network of interconnected queue managers, so that one queue manager can unambiguously identify the target queue manager to which any given message should be sent.

For the other types of objects, each object has a name associated with it and can be referenced by that name. These names must be unique within one queue manager and object type. For example, you can have a queue and a process with the same name, but you cannot have two queues with the same name.

In MQSeries, names can have a maximum of 48 characters, with the exception of *channels*, which have a maximum of 20 characters. For more information about names see “Rules for naming MQSeries objects” on page 225.

Managing objects

You can create, alter, display and delete objects using:

- Control commands, which are typed in from a keyboard
- MQSeries commands (MQSC), which can be typed in from a keyboard or read from a file.
- Programmable Command Format (PCF) commands, which can be used in an automation program.
- MQSeries Administration Interface (MQAI) calls in a program.

For more information, see “Chapter 2. An introduction to MQSeries administration” on page 17.

Object attributes

The properties of an object are defined by its attributes. Some you can specify, others you can only view. For example, the maximum message length that a queue can accommodate is defined by its *MaxMsgLength* attribute; you can specify this attribute when you create a queue. The *DefinitionType* attribute specifies how the queue was created; you can only display this attribute.

In MQSeries, there are two ways of referring to an attribute:

- Using its PCF name, for example, *MaxMsgLength*.
- Using its MQSC name, for example, MAXMSGL.

The formal name of an attribute is its PCF name. Because using the MQSC facility is an important part of this book, you are more likely to see the MQSC name in examples than the PCF name of a given attribute.

MQSeries queue managers

A queue manager provides queuing services to applications, and manages the queues that belong to it. It ensures that:

- Object attributes are changed according to the commands received.
- Special events such as trigger events or instrumentation events are generated when the appropriate conditions are met.

- Messages are put on the correct queue, as requested by the application making the MQPUT call. The application is informed if this cannot be done, and an appropriate reason code is given.

Each queue belongs to a single queue manager and is said to be a *local queue* to that queue manager. The queue manager to which an application is connected is said to be the local queue manager for that application. For the application, the queues that belong to its local queue manager are local queues.

A *remote queue* is simply a queue that belongs to another queue manager.

A *remote queue manager* is any queue manager other than the local queue manager. A remote queue manager may exist on a remote machine across the network or it may exist on the same machine as the local queue manager.

MQSeries supports multiple queue managers on the same machine.

MQI calls

A queue manager object may be used in some MQI calls. For example, you can inquire about the attributes of the queue manager object using the MQI call MQINQ.

Note: You cannot put messages on a queue manager object; messages are always put on queue objects, not on queue manager objects.

MQSeries queues

Queues are defined to MQSeries using:

- The appropriate MQSC DEFINE command
- The PCF Create Queue command

The commands specify the type of queue and its attributes. For example, a local queue object has attributes that specify what happens when applications reference that queue in MQI calls. Examples of attributes are:

- Whether applications can retrieve messages from the queue (GET enabled).
- Whether applications can put messages on the queue (PUT enabled).
- Whether access to the queue is exclusive to one application or shared between applications.
- The maximum number of messages that can be stored on the queue at the same time (maximum queue depth).
- The maximum length of messages that can be put on the queue.

For further details about defining queue objects, see the *MQSeries Command Reference* or the *MQSeries Programmable System Management* book.

Using queue objects

In MQSeries, there are various types of queue object. Each type of object can be manipulated by the product commands and is associated with real queues in different ways:

- **Local queue object**

  A local queue object identifies a local queue belonging to the queue manager to which the application is connected. All queues are local queues in the sense that each queue belongs to a queue manager and, for that queue manager, the queue is a local queue.

- **Remote queue object**

  A remote queue object identifies a queue belonging to another queue manager. This queue must be defined as a local queue to that queue manager. The information you specify when you define a remote queue object allows the local queue manager to find the remote queue manager, so that any messages destined for the remote queue go to the correct queue manager.

  Before applications can send messages to a queue on another queue manager, you must have defined a transmission queue and channels between the queue managers, unless you have grouped one or more queue managers together into a cluster. For more information about clusters, see "Remote administration using clusters" on page 60.
- **Alias queue object**

  An alias queue object allows applications to access a queue by referring to it indirectly in MQI calls. When an alias queue name is used in an MQI call, the name is resolved to the name of either a local or a remote queue at run time. This allows you to change the queues that applications use without changing the application in any way—you merely change the alias queue definition to reflect the name of the new queue to which the alias resolves.

  An alias queue is not a queue, but an object that you can use to access another queue.
- **Model queue object**

  A model queue object defines a set of queue attributes that are used as a template for creating a dynamic queue. Dynamic queues are created by the queue manager when an application issues an MQOPEN request specifying a queue name that is the name of a model queue. The dynamic queue that is created in this way is a local queue whose attributes are taken from the model queue definition. The dynamic queue name can be specified by the application or the queue manager can generate the name and return it to the application.

  Dynamic queues defined in this way may be temporary queues, which do not survive product restarts, or permanent queues, which do.

Specific local queues used by MQSeries

MQSeries uses some local queues for specific purposes related to its operation. You *must* define them before MQSeries can use them.

Application queues: A queue that is used by an application (through the MQI) is referred to as an *application queue.* This can be a local queue on the queue manager to which an application is connected, or it can be a remote queue that is owned by another queue manager.

Applications can put messages on local or remote queues. However, they can only get messages from a local queue.

Initiation queues: *Initiation queues* are queues that are used in triggering. A queue manager puts a trigger message on an initiation queue when a trigger event occurs. A trigger event is a logical combination of conditions that is detected by a queue manager. For example, a trigger event may be generated when the number of messages on a queue reaches a predefined depth. This event causes the queue manager to put a trigger message on a specified initiation queue. This trigger message is retrieved by a *trigger monitor*, a special application that monitors an initiation queue. The trigger monitor then starts up the application program that was specified in the trigger message.

If a queue manager is to use triggering, at least one initiation queue must be defined for that queue manager.

See "Managing objects for triggering" on page 51, and "runmqtrm (Start trigger monitor)" on page 280. For more information about triggering, see the *MQSeries Application Programming Guide*.

Transmission queues: A *transmission queue* temporarily stores messages that are destined for a remote queue manager. You must define at least one transmission queue for each remote queue manager to which the local queue manager is to send messages directly. These queues are also used in remote administration; see "Remote administration from a local queue manager using MQSC commands" on page 61. For information about the use of transmission queues in distributed queuing, see the *MQSeries Intercommunication* book.

Cluster transmission queues: Each queue manager within a cluster has a cluster transmission queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE. A definition of this queue is created by default on every queue manager on Version 5.1.

A queue manager that is part of the cluster can send messages on the cluster transmission queue to any other queue manager that is in the same cluster.

Cluster queue managers can communicate with queue managers that are not part of the cluster. To do this, you must define channels and a transmission queue from a queue manager within the cluster to the other queue manager outside the cluster in the same way as in a traditional distributed-queuing environment.

During name resolution, the cluster transmission queue takes precedence over the default transmission queue. When a queue manager that is not part of the cluster puts a message onto a remote queue, the default action, if there is no transmission queue with the same name as the destination queue manager, is to use the default transmission queue.

When a queue manager is part of a cluster, the default action is to use the SYSTEM.CLUSTER.TRANSMIT.QUEUE, except when the destination queue manager is not a part of the cluster.

Dead-letter queues: A *dead-letter queue* stores messages that cannot be routed to their correct destinations. This occurs when, for example, the destination queue is full. The supplied dead-letter queue is called SYSTEM.DEAD.LETTER.QUEUE. These queues are also referred to as undelivered-message queues on other platforms.

For distributed queuing, you should define a dead-letter queue on each queue manager involved.

Command queues: The command queue, named SYSTEM.ADMIN.COMMAND.QUEUE, is a local queue to which suitably authorized applications can send MQSeries commands for processing. These commands are then retrieved by an MQSeries component called the command server. The command server validates the commands, passes the valid ones on for processing by the queue manager, and returns any responses to the appropriate reply-to queue.

A command queue is created automatically for each queue manager when that queue manager is created.

Reply-to queues: When an application sends a request message, the application that receives the message can send back a reply message to the sending application. This message is put on a queue, called a reply-to queue, which is normally a local queue to the sending application. The name of the reply-to queue is specified by the sending application as part of the message descriptor.

Event queues: MQSeries Version 5.1 supports instrumentation events, which can be used to monitor queue managers independently of MQI applications. Instrumentation events can be generated in several ways, for example:

- An application attempting to put a message on a queue that is not available or does not exist.
- A queue becoming full.
- A channel being started.

When an instrumentation event occurs, the queue manager puts an event message on an event queue. This message can then be read by a monitoring application which may inform an administrator or initiate some remedial action if the event indicates a problem.

Note: Trigger events are quite different from instrumentation events in that trigger events are not caused by the same conditions, and do not generate event messages.

For more information about instrumentation events, see the *MQSeries Programmable System Management* book.

Process definitions

A *process definition object* defines an application that is to be started in response to a trigger event on an MQSeries queue manager. See “Initiation queues” on page 10 for more information.

The process definition attributes include the application ID, the application type, and data specific to the application.

Use the MQSC command DEFINE PROCESS or the PCF command Create Process to create a process definition.

Channels

Channels are objects that provide a communication path from one queue manager to another. Channels are used in distributed message queuing to move messages from one queue manager to another. They shield applications from the underlying communications protocols. The queue managers may exist on the same, or different, platforms. For queue managers to communicate with one another, you must define one channel object at the queue manager that is to send messages, and another, complementary one, at the queue manager that is to receive them.

For information on channels and how to use them, see the *MQSeries Intercommunication* book, and also “Preparing channels and transmission queues for remote administration” on page 62.

Clusters

In a traditional MQSeries network using distributed queuing, every queue manager is independent. If one queue manager needs to send messages to another queue manager it must have defined a transmission queue, a channel to the remote queue manager, and a remote queue definition for every queue to which it wants to send messages.

A cluster is a group of queue managers set up in such a way that the queue managers can communicate directly with one another over a single network without the need for complex transmission queues, channels and queue definitions.

Note: MQSeries clusters are not the same as OpenVMS clusters. When the term *cluster* is used, it refers to an MQSeries queue manager cluster. An OpenVMS cluster is always referred to as an *OpenVMS cluster*. For more on OpenVMS clusters, see "Chapter 16. MQSeries for OpenVMS and clustering" on page 203.

For information about clusters, see "Chapter 6. Administering remote MQSeries objects" on page 59 and the *MQSeries Queue Manager Clusters* book.

Namelists

A namelist is an MQSeries object that contains a list of other MQSeries objects. Typically, namelists are used by applications such as trigger monitors, where they are used to identify a group of queues. The advantage of using a namelist is that it is maintained independently of applications; that is, it can be updated without stopping any of the applications that use it. Also, if one application fails, the namelist is not affected and other applications can continue using it.

Namelists are also used with queue manager clusters so that you can maintain a list of clusters referenced by more than one MQSeries object.

System default objects

The *system default objects* are a set of object definitions that are created automatically whenever a queue manager is created. You can copy and modify any of these object definitions for use in applications at your installation. Default object names have the stem SYSTEM.DEF; for example, the default local queue is SYSTEM.DEFAULT.LOCAL.QUEUE; the default receiver channel is SYSTEM.DEF.RECEIVER. You cannot rename these objects; default objects of these names are required.

When you define an object, any attributes that you do not specify explicitly are copied from the appropriate default object. For example, if you define a local queue, the attributes you do not specify are taken from the default queue SYSTEM.DEFAULT.LOCAL.QUEUE.

See "Appendix B. System defaults" on page 297 for more information about system defaults.

Local and remote administration

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

Local and remote administration

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

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

Clients and servers

MQSeries supports client-server configurations for MQSeries applications.

An *MQSeries client* is a part of the MQSeries product that is installed on a machine to accept MQI calls from applications and pass them to an *MQI server* machine. There they are processed by a queue manager. Typically, the client and server reside on different machines but they can also exist on the same machine.

An *MQI server* is a queue manager that provides queuing services to one or more clients. All the MQSeries objects, for example queues, exist only on the queue manager machine, that is, on the MQI server machine. A server can support normal local MQSeries applications as well.

The difference between an MQI server and an ordinary queue manager is that a server has a dedicated communications link with each client. For more information about creating channels for clients and servers, see the *MQSeries Intercommunication* book.

For information about client support in general, see the *MQSeries Clients* book.

MQSeries applications in a client-server environment

When linked to a server, client MQSeries applications can issue MQI calls in the same way as local applications. The client application issues an MQCONN call to connect to a specified queue manager. Any additional MQI calls that specify the connection handle returned from the connect request are then processed by this queue manager.

You must link your applications to the appropriate client libraries. See the *MQSeries Clients* book for further information.

Extending queue manager facilities

The facilities provided by a queue manager can be extended by:
- User exits
- Installable services

User exits

User exits provide a mechanism for users to insert their own code into a queue manager function. The supported user exits are:

- **Channel exits**

  These exits change the way that channels operate. Channel exits are described in the *MQSeries Intercommunication* book.
- **Data conversion exits**

  These exits create source code fragments that can be put into application programs to convert data from one format to another. Data conversion exits are described in the *MQSeries Application Programming Guide*.
- **Cluster workload exit**

  The function performed by this exit is defined by the provider of the exit. Call definition information is given in the *MQSeries Queue Manager Clusters* book.

All types of exit are related to distributed queueing. For more information about these exits and how to use them, see the *MQSeries Intercommunication* book.

Installable services

Installable services are more extensive than exits in that they have formalized interfaces (an API) with multiple entry points.

An implementation of an installable service is called a *service component*. You can use the components supplied with the product, or you can write your own component to perform the functions that you require.

Currently, the following installable services are provided:

- **Authorization service**.

  The authorization service allows you to build your own security facility.

  The default service component that implements the service is the Object Authority Manager (OAM), which is supplied with the product. By default, the OAM is active, that is, you do not have to do anything to configure it. You can use the authorization service interface to create other components to replace or augment the OAM. For more information about the OAM, see “Chapter 7. Protecting MQSeries objects” on page 73.
- **Name service**.

  The name service enables the sharing of queues by allowing applications to identify remote queues as though they were local queues. A default service component that implements the name service is provided with MQSeries Version 5.1. It uses the Open Software Foundation (OSF) Distributed Computing Environment (DCE). You can also write your own name service component, for example, if you do not have DCE installed. By default, the name service is inactive.

See “Chapter 12. Using the name service” on page 157 and also the *MQSeries Programmable System Management* book.

Security

In the MQSeries Version 5 products, there are two methods of providing security:

- The Object Authority Manager (OAM) facility
- DCE security

Object Authority manager (OAM) facility

Authorization for using MQI calls, commands, and access to objects is provided by the Object Authority Manager (OAM), which by default is enabled. Access to MQSeries entities is controlled through MQSeries user groups and the OAM. A command line interface is provided to enable administrators to grant or revoke authorizations as required.

For more information about creating authorization service components, see the *MQSeries Programmable System Management* book.

DCE security

Channel exits that use the DCE Generic Security Service (GSS) are provided by MQSeries. For more information, see the *MQSeries Intercommunication* book.

Transactional support

An application program can group a set of updates into a *unit of work*. These updates are usually logically related and must all be successful for data integrity to be preserved. If one update succeeded while another failed then data integrity would be lost.

A unit of work commits when it completes successfully. At this point all updates made within that unit of work are made permanent or irreversible. If the unit of work fails then all updates are instead backed out. Syncpoint coordination is the process by which units of work are either committed or backed out with integrity.

A *local* unit of work is one in which the only resources updated are those of the MQSeries queue manager. Here syncpoint coordination is provided by the queue manager itself using a single-phase commit process.

A *global* unit of work is one in which resources belonging to other resource managers, such as XA-compliant databases, are also updated. Here, a two-phase commit procedure must be used and the unit of work may be coordinated by the queue manager itself.

For more information, see “Chapter 10. Transactional support” on page 111.

Chapter 2. An introduction to MQSeries administration

This chapter introduces the subject of MQSeries administration.

Administration tasks include creating, starting, altering, viewing, stopping, and deleting MQSeries objects (queue managers, queues, processes, namelists, clusters, and channels).

This chapter contains the following sections:

Local and remote administration

You administer MQSeries objects locally or remotely.

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

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

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

"Chapter 6. Administering remote MQSeries objects" on page 59 describes the subject of remote administration in greater detail.

Performing administration tasks using control commands

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

See "Chapter 3. Managing queue managers using control commands" on page 23 for more information about control commands.

Performing administrative tasks using MQSC commands

You use the MQSeries commands (MQSC) to manage queue manager objects, including the queue manager itself, channels, queues, and process definitions. For example, there are commands to define, alter, display, and delete a specified queue.

You run MQSC commands by invoking the control command **runmqsc** from a command line. You can run MQSC commands:

- Interactively by typing them at the keyboard. See "Using the MQSC facility interactively" on page 33.
- As a sequence of commands from an ASCII text file. See "Running MQSC commands from text files" on page 36.

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

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

For more information about using the MQSC facility and text files, see "Running MQSC commands from text files" on page 36. For more information about the **runmqsc** command, see "runmqsc (Run MQSeries commands)" on page 276.

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

MQSC commands are available on other platforms, including AS/400® and OS/390.

MQSC commands are summarized in "Appendix D. Comparing command sets" on page 305.

See the *MQSeries Command Reference* book for a description of each MQSC command and its syntax.

See "Performing local administration tasks using MQSC commands" on page 32 for more information about using MQSC commands in local administration.

Performing administrative tasks using PCF commands

The purpose of the MQSeries programmable command format (PCF) commands is to allow administration tasks to be programmed into an administration program. In this way you can create queues and process definitions, and change queue managers, from a program.

PCF commands cover the same range of functions that are provided by the MQSC facility.

See "PCF commands" on page 55 for more information.

For a complete description of the PCF data structures and how to implement them, see the *MQSeries Programmable System Management* book.

You can use the MQSeries Administration Interface (MQAI) to obtain easier programming access to PCF messages. This is described in greater detail in "Using the MQAI to simplify the use of PCFs" on page 56.

Attributes in MQSC and PCFs

Object attributes specified in MQSC are shown in this book in uppercase, for example RQMNAME, although they are not case sensitive. These attribute names are limited to eight characters, so it is not easy to work out the meaning of some of them, for example, QDPHIEV. Object attributes in PCF are shown in italics, are not limited to eight characters, and are therefore easier to read. The PCF equivalent of RQMNAME, is *RemoteQMgrName* and of QDPHIEV is *QDepthHighEvent*.

Escape PCFs

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

Understanding MQSeries file names

Each MQSeries queue, queue manager, namelist, and process object is represented by a file. Because object names are not necessarily valid file names, the queue manager converts the object name into a valid file name where necessary.

The path to a queue manager directory is formed from the following:

- A prefix - the first part of the name:
  ```
  MQS_ROOT:[MQM]
  ```
  This prefix is defined in the queue manager configuration file.
- A literal:
  ```
  QMGRS
  ```
- A coded queue manager name, which is the queue manager name transformed into a valid directory name. For example, the queue manager:
  ```
  QUEUE.MANAGER
  ```
  would be represented as:
  ```
  QUEUE$MANAGER
  ```
  This process is referred to as *name transformation*.

Queue manager name transformation

In MQSeries you can give a queue manager a name containing up to 48 characters.

For example, you could name a queue manager:

```
QUEUE.MANAGER.ACCOUNTING.SERVICES
```

However, each queue manager is represented by a file and there are limitations to the maximum length a file name can be, and to the characters that can be used in the name. As a result, the names of files representing objects are automatically transformed to meet the requirements of the file system.

Understanding MQSeries names

The rules governing the transformation of a queue manager name, using the example of a queue manager with the name `QUEUE.MANAGER`, are as follows:

1. Transform individual characters:
   · becomes $
   / becomes _
   % becomes _
2. If the name is still not valid:
   a. Truncate it to eight characters
   b. Append a three-character numeric suffix

For example, assuming the default prefix, the queue manager name becomes:

```
MQS_ROOT:[MQM.QMGRS.QUEUE$MANAGER]
```

The transformation algorithm also allows distinction between names that differ only in case, on file systems that are not case sensitive.

Object name transformation

Object names are not necessarily valid file system names. Therefore the object names may need to be transformed. The method used is different from that for queue manager names because, although there only a few queue manager names per machine, there can be a large number of other objects for each queue manager. Only process definitions, queues and namelists are represented in the file system; channels are not affected by these considerations.

When a new name is generated by the transformation process there is no simple relationship with the original object name. You can use the **dspmqfls** command to convert between real and transformed object names.

Queue file names begin with the letter “Q”.

For more information on naming objects, see “Rules for naming MQSeries objects” on page 225.

Understanding case sensitivity

Case sensitivity in control commands

OpenVMS is normally described as a case-insensitive operating system. This means that, in general, the following three commands all create a queue manager called ″QUEUEMANAGER″.

```
$ crtmqm QueueManager
$ crtmqm queuemanager
$ crtmqm QUEUEMANAGER
```

With MQSeries for Compaq OpenVMS, you can use double quotes around the name of the queue manager (or similar parameter) to protect its case. When the double quotes are used, the following three commands now create three different queue managers.

```
$ crtmqm "QueueManager"   creates a queue manager called QueueManager
$ crtmqm "queuemanager"   creates a queue manager called queuemanager
$ crtmqm "QUEUEMANAGER"   creates a queue manager called QUEUEMANAGER
```

OpenVMS Version 7.2 introduced the following new command:

```
$ set process /parse_style = ( traditional | extended )
```

This command changes how OpenVMS handles upper and lower case characters.

If **set process /parse_style** command is not used, or if it is used with the **traditional** option, then OpenVMS behaves as it always has done with regard to case sensitivity.

If this command is used with the **extended** option, the behaviour of the LIB$GET_FOREIGN run time library routine changes so that it preserves the case of text that it retrieves. Because MQSeries uses this routine to obtain command line parameters, the case of the parameters is preserved, even when the parameters are not enclosed in double quotes.

For example, the following sequence of commands creates three different queue managers. Notice the parameters are not enclosed in double quotes.

```
$ set process /parse_style = extended
$ crtmqm QueueManager   creates a queue manager called QueueManager
$ crtmqm queuemanager   creates a queue manager called queuemanager
$ crtmqm QUEUEMANAGER   creates a queue manager called QUEUEMANAGER
```

The OpenVMS set process /parse_style command changes a number of things besides case sensitivity. You may want to learn more about the command from the information provided in the OpenVMS DCL Dictionary before applying it to your system.

Case sensitivity in MQSC commands

MQSeries control commands (for example, **runmqsc** which invokes the MQSC facility) are not case sensitive.

MQSC commands, including their attributes, can be written in upper or lower case. Object names in MQSC commands are automatically converted to upper case unless the names are enclosed in *single* quotation marks. If single quotation marks are not used, the object is processed with a name in upper case. See the *MQSeries Command Reference* book for more information.

Chapter 3. Managing queue managers using control commands

This chapter describes how you can perform operations on queue managers and command servers. It contains these sections:

- "Using control commands"
- "Guidelines for creating queue managers" on page 24
- "Creating a default queue manager" on page 27
- "Starting a queue manager" on page 28
- "Making an existing queue manager the default" on page 28
- "Stopping a queue manager" on page 28
- "Restarting a queue manager" on page 30
- "Deleting a queue manager" on page 30
- "Looking at object files" on page 225

Using control commands

You use control commands to perform operations on queue managers, command servers, and channels. Control commands can be divided into three categories, as shown in Table 1 on page 23.

Table 1. Categories of control commands

Category	Description
Queue manager commands	Queue manager control commands include commands for creating, starting, stopping, and deleting queue managers and command servers.
Channel commands	Channel commands include commands for starting and ending channels and channel initiators.
Utility commands	Utility commands include commands associated with: • Running MQSC commands • Conversion exits • Authority management • Recording and recovering media images of queue manager resources • Displaying and resolving transactions • Trigger monitors • Displaying the file names of MQSeries objects

For information about administration tasks for channels, see the *MQSeries Intercommunication* book.

Using control commands

In MQSeries for Compaq OpenVMS, you enter control commands at a DCL prompt. The command name and flags themselves are not case sensitive, but the parameters may or may not be converted to upper case, depending on an OpenVMS process option and whether the parameters were enclosed in double quotes to protect the case. For more on how the OpenVMS command and double quotes affect case, see “Understanding case sensitivity” on page 20.

Typically, in this example:

```
crtmqm -u system.dead.letter.queue "jupiter.queue.manager"
```

- The dead-letter queue could be SYSTEM.DEAD.LETTER.QUEUE, even though it was entered in lower case. Whether its case is automatically converted to upper case depends on the setting of the OpenVMS command **set process/parse_style**. See “Understanding case sensitivity” on page 20.
- The queue manager name is specified as ″jupiter.queue.manager″ (which is different from ″JUPITER.queue.manager″) because it was enclosed in double quotes.

Therefore, take care to type the commands exactly as you see them in the examples.

Creating a queue manager

A queue manager manages the resources associated with it, in particular the queues that it owns. It provides queuing services to applications for Message Queueing Interface (MQI) calls and commands to create, modify, display, and delete MQSeries objects.

Before you can do anything with messages and queues, you must create at least one queue manager and its associated objects. To create a queue manager, you use the MQSeries control command **crtmqm**. The **crtmqm** command automatically creates the required default objects and system objects. Default objects form the basis of any object definitions that you make; system objects are required for queue manager operation. When a queue manger and its objects have been created, you use the **strmqm** command to start the queue manager.

Guidelines for creating queue managers

Before creating a queue manager, there are several points you need to consider (especially in a production environment). Work through this checklist:

- Specifying a unique queue manager name.
- Limiting the number of queue managers.
- Specifying a default queue manager.
- Specifying a dead-letter queue.
- Specifying a default transmission queue.
- Specifying the required logging parameters.
- Backing up configuration files after creating a queue manager.

The tasks in this list are explained in the sections that follow.

Specifying a unique queue manager name

When you create a queue manager, ensure that no other queue manager has the same name *anywhere* in your network. Queue manager names are not checked at creation time, and names that are not unique will prevent you from using channels for distributed queuing.

One way of ensuring uniqueness is to prefix each queue manager name with its own (unique) node name. For example, if a node is called `accounts`, you could name your queue manager `accounts.saturn.queue.manager`, where `saturn` identifies a particular queue manager and `queue.manager` is an extension you can give to all queue managers. Alternatively, you can omit this, but note that `accounts.saturn` and `accounts.saturn.queue.manager` are *different* queue manager names.

If you are using MQSeries for communicating with other enterprises, you can also include your own enterprise as a prefix. We do not actually do this in the examples, because it makes them more difficult to follow.

Note: Queue manager names in control commands may or may not be converted to upper case, depending on an OpenVMS process option and whether the queue manager name was enclosed in double quotes to protect the case. This means that you could create two queue managers with the names `jupiter.queue.manager` and `JUPITER.queue.manager`. For more on how the OpenVMS process option and double quotes affect case, see “Understanding case sensitivity” on page 20.

Limiting the number of queue managers

You can create as many queue managers as resources allow. However, because each queue manager requires its own resources, it is generally better to have one queue manager with 100 queues on a node than to have ten queue managers with ten queues each.

In production systems, many nodes will be run with a single queue manager, but larger server machines may run with multiple queue managers.

Specifying the default queue manager

Each node should have a default queue manager, though it is possible to configure MQSeries on a node without one.

To create a queue manager use the **crtmqm** command. For a detailed description of this command and its parameters, see “crtmqm (Create queue manager)” on page 231.

What is a default queue manager?

The default queue manager is the queue manager that applications connect to if they do not specify a queue manager name in an MQCONN call. It is also the queue manager that processes MQSC commands when you invoke the **runmqsc** command without specifying a queue manager name.

How do you specify a default queue manager?

You include the -q flag on the **crtmqm** command to specify that the queue manager you are creating is the default queue manager. Omit this flag if you do not want to the queue manager you are creating to become a default queue manager.

Specifying a queue manager as the default *replaces* any existing default queue manager specification for the node.

What happens if I decide to change the default queue manager?

If you decide to change the default queue manager, be aware that this can affect other users or applications. The change has no effect on currently-connected applications, because they can use the handle from their original connect call in any further MQI calls. This handle ensures that the calls are directed to the same queue manager. Any applications connecting after the change connect to the new default queue manager.

This may be what you intend, but you should take this into account before you change the default.

Specifying a dead-letter queue

The dead-letter queue is a local queue where messages are put if they cannot be routed to their correct destination.

Attention:
It is vitally important to have a dead-letter queue on each queue manager in your network. Failure to do so may mean that errors in application programs cause channels to be closed or that replies to administration commands are not received.

For example, if an application attempts to put a message on a queue on another queue manager, but the wrong queue name is given, the channel is stopped, and the message remains on the transmission queue. Other applications cannot then use this channel for their messages.

The channels are not affected if the queue managers have dead-letter queues. The undelivered message is simply put on the dead-letter queue at the receiving end, leaving the channel and its transmission queue available.

Therefore, when you create a queue manager you should use the -u flag to specify the name of the dead-letter queue. You can also use an MQSC command to alter the attributes of a queue manager and specify the dead-letter queue to be used. See "Altering queue manager attributes" on page 35 for an example of an MQSC ALTER command.

When you find messages on a dead-letter queue, you can use the dead-letter queue handler, supplied with MQSeries, to process these messages. See "Chapter 8. The MQSeries dead-letter queue handler" on page 93 for further information about the dead-letter queue handler itself, and how to reduce the number of messages that might otherwise be placed on the dead-letter queue.

Specifying a default transmission queue

A transmission queue is a local queue on which messages in transit to a remote queue manager are queued pending transmission. The default transmission queue is the queue that is used when no transmission queue is explicitly defined. Each queue manager can be assigned a default transmission queue.

When you create a queue manager you should use the -d flag to specify the name of the default transmission queue. This does not actually create the queue; you have to do this explicitly later on. See "Working with local queues" on page 40 for more information.

Specifying the required logging parameters

You can specify logging parameters on the **crtmqm** command, including the type of logging, and the path and size of the log files. In a development environment, the default logging parameters should be adequate. However, you can change the defaults if, for example:

- You have a low-end system configuration that cannot support large logs.

- You anticipate a large number of long messages being on your queues at the same time.

For more information about specifying logging parameters:
- On the **crtmqm** command, see "crtmqm (Create queue manager)" on page 231.
- Using configuration files, see "The Log stanza" on page 167.

Backing up configuration files after creating a queue manager

There are two configuration files to consider:
1. When you install the product, the MQSeries configuration file (mqs.ini) is created. It contains a list of queue managers, which is updated each time you create or delete a queue manager. There is one mqs.ini file per node.
2. When you create a new queue manager, a new queue manager configuration file (qm.ini) is automatically created. This contains configuration parameters for the queue manager.

You should make a backup of these files. If, later on, you create another queue manager that causes you problems, you can reinstate the backups when you have removed the source of the problem. As a general rule, you should back up your configuration files each time you create a new queue manager.

For more information about configuration files, see "Chapter 13. Configuring MQSeries" on page 159.

Creating a default queue manager

You create a default queue manager using the **crtmqm** command. The **crtmqm** command specified with a q flag:
- Creates a default queue manager called `saturn.queue.manager`
- Creates the default and system objects
- Specifies the names of both its default transmission queue and its dead-letter queue

```
crtmqm -q -d MY.DEFAULT.XMIT.QUEUE -u SYSTEM.DEAD.LETTER.QUEUE "saturn.queue.manager"
```

where:

-q Indicates that this queue manager is the default queue manager.

-d MY.DEFAULT.XMIT.QUEUE
Is the name of the default transmission queue.

-u SYSTEM.DEAD.LETTER.QUEUE
Is the name of the dead-letter queue.

"saturn.queue.manager"
Is the name of this queue manager. For **crtmqm**, this must be the last parameter in the command.

Creating a default queue manager allows you to issue some commands against it (such as **strmqm** and **runmqsc**) without having to specify a queue manager name. Other commands (such as **endmqm** and **dltmqm**) require a specified queue manager name.

Notice that the queue manager name in this example is in lower case and that the the lower case is protected by double quotes. For more information on how case sensitivity is handled for parameters, see “Understanding case sensitivity” on page 20.

Starting a queue manager

Although you have created a queue manager, it cannot process commands or MQI calls until it has been started. Start the queue manager by typing in this command:

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

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

Making an existing queue manager the default

When you create a default queue manager, the name of the default queue manager is inserted in the *DefaultQueueManager* stanza in the MQSeries configuration file (mqs.ini). The stanza and its contents are automatically created if they do not exist.

You may need to edit this stanza:

- **To make an existing queue manager the default.** To do this you have to change the queue manager name in this stanza to the name of the new default queue manager. You must do this manually, using a text editor.
- **If you do not have a default queue manager on the node, and you want to make an existing queue manager the default.** To do this you must create the *DefaultQueueManager* stanza—with the required name—yourself.
- **If you accidentally make another queue manager the default and wish to revert to the original default queue manager.** To do this, edit the *DefaultQueueManager* stanza in the MQSeries configuration file, replacing the name of the unwanted default queue manager with that of the one you do want.

See “Chapter 13. Configuring MQSeries” on page 159 for information about configuration files.

When the stanza contains the required information, stop the queue manager and restart it.

Stopping a queue manager

You use the **endmqm** command to stop a queue manager. For example, to stop a queue manager called `saturn.queue.manager` type:

```
endmqm "saturn.queue.manager"
```

Quiesced shutdown

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

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

```
endmqm -c "saturn.queue.manager"
```

you are not told when all applications have stopped. (An endmqm -c "saturn.queue.manager" command is equivalent to an endmqm "saturn.queue.manager" command.)

Immediate shutdown

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

Use this as a normal way to stop the queue manager, optionally after a quiesce period. For an immediate shutdown, type:

```
endmqm -i "saturn.queue.manager"
```

Preemptive shutdown

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

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

```
endmqm -p "saturn.queue.manager"
```

This stops all queue manager code immediately.

Note: After a forced or preemptive shutdown, or if the queue manager fails, the queue manager may have ended without cleaning up the shared memory that it owns. This can lead to problems restarting. For information on how to use the MONMQ utility to clean up after an abrupt ending of this type, see "Managing shared memory with MONMQ" on page 342.

If you have problems shutting down a queue manager

Problems in shutting down a queue manager are often caused by applications. For example, when applications:
- Do not check MQI return codes properly.
- Do not request a notification of a quiesce.
- Terminate without disconnecting from the queue manager (by issuing an MQDISC call).

If a shutdown of a queue manager is very slow, or you believe that the queue manager is not going to stop, you can break out of the **endmqm** command using Ctrl-Y. You can then issue another **endmqm** command, but this time with a flag specifying either an immediate or a preemptive shutdown.

For a detailed description of the **endmqm** command and its options, see "endmqm (End queue manager)" on page 254.

Restarting a queue manager

To restart a queue manager, use the command:

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

Deleting a queue manager

To delete a queue manager, first stop it, then use the following command:

```
dltmqm "saturn.queue.manager"
```

Attention: Deleting a queue manager is a drastic step, because you also delete all the resources associated with it. This includes not only all queues and their messages, but also all object definitions.

For a description of the **dltmqm** command and its options, see "dltmqm (Delete queue manager)" on page 235. You should ensure that only trusted administrators have the authority to use this command.

Chapter 4. Administering local MQSeries objects

This chapter describes how to administer local MQSeries objects to support application programs that use the Message Queuing Interface (MQI). In this context, local administration means creating, displaying, changing, copying, and deleting MQSeries objects.

This chapter contains these sections:

Supporting application programs that use the MQI

MQSeries application programs need certain objects before they can run successfully. For example, Figure 1 shows an application that removes messages from a queue, processes them, and then sends some results to another queue on the same queue manager.

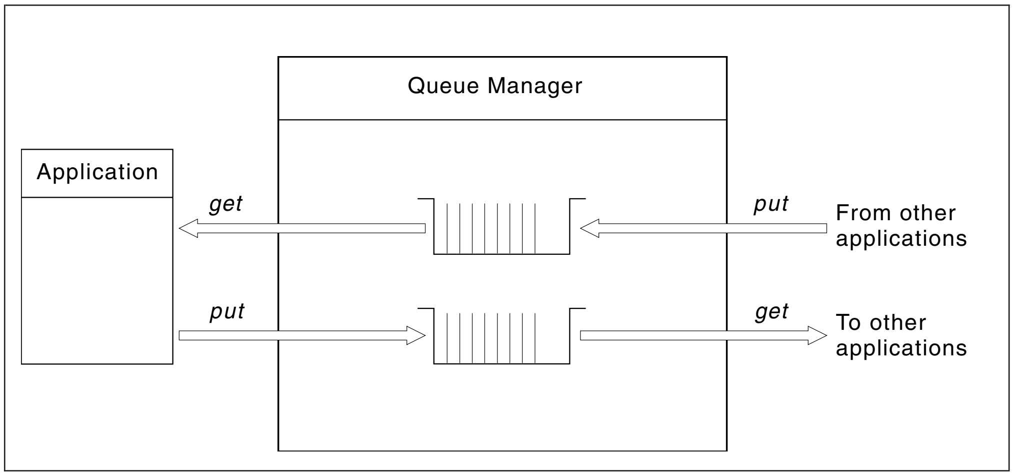


Figure 1. Queues, messages, and applications

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

Before this application can be run, these conditions must be satisfied:

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

Application programs

- The application must be able to connect to the queue manager. To do this it must be linked to the product code. See the *MQSeries Application Programming Guide* for more information.
- The applications that put the messages on the first queue must also connect to a queue manager. If they are remote, they must also be set up with transmission queues and channels. This part of the system is not shown in Figure 1 on page 31.

Performing local administration tasks using MQSC commands

In this section, we assume that you will be issuing commands using the **runmqsc** command. You can do this interactively—entering the commands at the keyboard—or you can redirect SYS$INPUT to run a sequence of commands from an ASCII text file. In both cases, the format of the commands is the same.

The *MQSeries Command Reference* book contains a description of each MQSC command and its syntax.

You can use MQSeries script commands (MQSC) to manage queue manger objects, including the queue manager itself, clusters, channels, queues, namelists and process definitions. This section deals with queue managers, queues and process definitions; for information about administering channel objects, see DQM implementation in the *MQSeries Intercommunication* book.

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

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

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

Object attributes specified in MQSC are shown in this book in upper case (for example, RQMNAME) although they are not case sensitive. (For more on case sensitivity, see “Case sensitivity in MQSC commands” on page 21.) MQSC attribute names are limited to eight characters.

Before you start

Before you can run MQSC commands, you must have created and started the queue manager that is going to run the commands, see “Creating a default queue manager” on page 27.

MQSeries object names

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

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

```
ORANGE.LOCAL.QUEUE
```

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

We also use the name `saturn.queue.manager` as a queue manager name.

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

You do not have to use these names, but if you do not, you must modify any commands in examples that specify them.

Redirecting input and output

To improve the ease of migration from other operating systems to OpenVMS, MQSeries supports the UNIX® style of redirection indicators for sys$input, sys$output, and sys$error, as follows:

< specifies the source for SYS$INPUT
> specifies the source for SYS$OUTPUT
2> specifies the source for SYS$ERROR

This feature is also included with the executable versions of the sample programs. However, it is not included in the source of the samples and, therefore, will not be available if you rebuild the samples from the source code.

Using the MQSC facility interactively

To enter commands interactively, at a DCL prompt type:

```
runmqsc
```

In this command, a queue manager name has not been specified, therefore the MQSC commands will be processed by the default queue manager. Now you can type in any MQSC commands, as required. For example, try this one:

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

Continuation characters must be used to indicate that a command is continued on the following line:

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

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

Feedback from MQSC commands

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

```
AMQ8006: MQSeries queue created
 .
 .
 .
AMQ8405: Syntax error detected at or near end of command segment below:-
z

AMQ8426: Valid MQSC commands are:

    ALTER
    CLEAR
    DEFINE
    DELETE
    DISPLAY
    END
    PING
    REFRESH
    RESET
    RESOLVE
    RESUME
    START
    STOP
    SUSPEND
```

The first message confirms that a queue has been created; the second indicates that you have made a syntax error. These messages are sent to the standard output device. If you have not entered the command correctly, refer to the *MQSeries Command Reference* book for the correct syntax.

Ending interactive input to MQSC

To end interactive input of MQSC commands, type the MQSC END command:

```
END
```

Alternatively, you can exit by typing the EOF character <CTRL Z>.

If you are redirecting input from other sources, such as a text file, you do not have to do this.

Displaying queue manager attributes

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

```
DISPLAY QMGR ALL
```

A typical output is displayed in Figure 2 on page 35.

```
  1 : display qmgr all
AMQ8408: Display Queue Manager details.
   DESCR( )                               DEADQ( )
   DEFXMITQ( )                            CHADEXIT( )
   CLWLEXIT( )                            CLWLDATA( )
   REPOS( )                               REPOSNL( )
   COMMANDQ(SYSTEM.ADMIN.COMMAND.QUEUE)   QMNAME(saturn.queue.manager)
   CRDATE(2001-01-16)                     CRTIME(11.13.56)
   ALTDATE(2001-01-16)                    ALTTIME(11.13.56)
   QMID(saturn.queue.manager_2001-01-16_11.13.56)
   TRIGINT(999999999)                     MAXHANDS(256)
   MAXUMSGS(10000)                        AUTHOREV(DISABLED)
   INHIBTEV(DISABLED)                     LOCALEV(DISABLED)
   REMOTEEV(DISABLED)                     PERFMEV(DISABLED)
   STRSTPEV(ENABLED)                      CHAD(DISABLED)
   CHADEV(DISABLED)                       CLWLLEN(100)
   MAXMSGL(4194304)                       CCSID(819)
   MAXPRTY(9)                             CMDLEVEL(510)
   PLATFORM(OpenVMS)                      SYNCPT
   DISTL(YES)
```

Figure 2. Typical output from a DISPLAY QMGR command

The ALL parameter on the DISPLAY QMGR command causes all the queue manager attributes to be displayed. In particular, because no queue manager name was specified when the command was run, the output tells us the default queue manager name (`saturn.queue.manager`), and the names of the dead-letter queue (SYSTEM.DEAD.LETTER.QUEUE) and the command queue (SYSTEM.ADMIN.COMMAND.QUEUE).

Before you go further, confirm that these queues have been created by typing the command:

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

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

Using a queue manager that is not the default

You can specify the queue manager name on the **runmqsc** command to run MQSC commands on a local queue manager other than the default. For example, to run MQSC commands on queue manager `jupiter.queue.manager`, use the command:

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

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

You can also run MQSC commands on a remote queue manager; see "Issuing MQSC commands remotely" on page 65.

Altering queue manager attributes

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

```
runmqsc "jupiter.queue.manager"

ALTER QMGR DEADQ (ANOTHERDLQ) INHIBTEV (ENABLED)
```

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

Running MQSC commands from text files

Running MQSC commands interactively is suitable for quick tests, but if you have very long commands, or sequences of commands that you want to repeat, you should provide input from a text file. (See "Redirecting input and output" on page 33 for information about redirection indicators.) To do this, first create a text file containing the MQSC commands using your familiar text editor. For example, the following command runs a sequence of commands contained in the text file `myprog.in`:

```
runmqsc < myprog.in
```

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

To redirect both `SYS$INPUT` and `SYS$OUTPUT` on the **runmqsc** command, use this form of the command:

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

This command invokes the MQSC commands contained in the MQSC command file `myprog.in`. Because we have not specified a queue manager name, the MQSC commands are run against the default queue manager. The output is sent to the report file `myprog.out`. Figure 3 on page 37 shows an extract from the MQSC command file `myprog.in` and Figure 4 on page 38 shows the corresponding extract of the report file `myprog.out`.

To redirect `SYS$INPUT` and `SYS$OUTPUT` on the **runmqsc** command, for a queue manager (`saturn.queue.manager`) that is not the default, use this form of the command:

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

MQSC command files

MQSC commands are written in human-readable form, that is, in ASCII text. Figure 3 on page 37 is an extract from an MQSC command file showing an MQSC command (DEFINE QLOCAL) with its attributes. The *MQSeries Command Reference* book contains a description of each MQSC command and its syntax.

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

Figure 3. Extract from the MQSC command file, myprog.in

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

For MQSeries for Compaq OpenVMS, you must limit lines to a maximum of 80 characters, including the continuation character. The plus sign indicates that the command is continued on the next line.

MQSC reports

The **runmqsc** command returns a *report*, which is sent to SYS$OUTPUT. The report contains:

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

  ```
  Starting MQSeries Commands.
  ```
- An optional numbered listing of the MQSC commands issued. By default, the text of the input is echoed to the output. Within this output, each command is prefixed by a sequence number, as shown in Figure 4 on page 38. However, you can use the -e flag on the **runmqsc** command to suppress the output.
- A syntax error message for any commands found to be in error.
- An *operator message* indicating the outcome of running each command. For example, the operator message for the successful completion of a DEFINE QLOCAL command is:

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

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

```
Starting MQSeries Commands.
 .
 .
     12:     DEFINE QLOCAL('RED.LOCAL.QUEUE') REPLACE  +
       :            DESCR(' ') +
       :            PUT(ENABLED) +
       :            DEFPRTY(0) +
       :            DEFPSIST(NO) +
       :            GET(ENABLED) +
       :            MAXDEPTH(5000) +
       :            MAXMSGL(1024) +
       :            DEFSOPT(SHARED) +
       :            USAGE(NORMAL) +
       :            NOTRIGGER
AMQ8006: MQSeries queue created.
       :
 .
 .
15 MQSC commands read.
0 commands have a syntax error.
0 commands cannot be processed.
```

Figure 4. Extract from the MQSC report file, myprog.out.

Running the supplied MQSC command files

When you install MQSeries for Compaq OpenVMS, the following MQSC command files is supplied:

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

The file is located in the directory `MQS_EXAMPLES:`

Using runmqsc to verify commands

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

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

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

This report is similar to that shown in Figure 4.

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

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

the -w flag, which you use to indicate that the queue manager is remote, is ignored, and the command is run locally in verification mode .

Resolving problems with MQSC

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

When you use the **runmqsc** command, remember:

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

```
AMQ8118: MQSeries queue manager does not exist.
```

- If you redirect output to a file, use the **>** indirection operator. By default, the output goes to the directory from which you ran the **runmqsc** command. Specify a fully-qualified file name to send your output to a specific file and directory.
- Check that you have created the queue manager that is going to run the commands.

  To do this, look in the configuration file mqs.ini, which by default is located in the MQS_ROOT:[MQM] directory. This file contains the names of the queue managers and the name of the default queue manager, if you have one.
- The queue manager should already be started. If it is not, start it; see “Starting a queue manager” on page 28. You get an error message if it is already started.
- Specify a queue manager name on the **runmqsc** command if you have not defined a default queue manager, otherwise you get this error:

```
AMQ8146: MQSeries queue manager not available.
```

  To correct this type of problem, see “Making an existing queue manager the default” on page 28.
- You cannot specify an MQSC command as a **runmqsc** parameter. For example, the following is invalid:

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

- You cannot enter MQSC commands from DCL before you issue the **runmqsc** command. For example:

```
DEFINE QLOCAL(QUEUE1)

%DCL-W-PARMDEL, invalid parameter delimiter - check use of special characters
```

Problems with MQSC

- You cannot run control commands from **runmqsc**. For example, you cannot start a queue manager once you are running MQSC interactively:

```
$ runmqsc
0790997, 5724-A38 (C) Copyright IBM Corp. 1996, 2001  ALL RIGHTS RESERVED.
Starting MQSeries Commands.


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

AMQ8426: Valid MQSC commands are:

    ALTER
    CLEAR
    DEFINE
    DELETE
    DISPLAY
    END
    PING
    REFRESH
    RESET
    RESOLVE
    RESUME
    START
    STOP
    SUSPEND

*CANCEL*


One MQSC command read.
One command has a syntax error.
All valid MQSC commands were processed.
$
```

See also “If you have problems using MQSC remotely” on page 67.

Working with local queues

This section contains examples of some of the MQSC commands that you can use. Refer to the *MQSeries Command Reference* book for a complete description of these commands.

Defining a local queue

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

Use the MQSC command DEFINE QLOCAL to create a definition of a local queue and also to create the data structure that is called a queue. You can also modify the queue characteristics from those of the default local queue.

In this example, the queue we define, ORANGE.LOCAL.QUEUE, is specified to have these characteristics:

- It is enabled for gets, disabled for puts, and operates on a first-in-first-out (FIFO) basis.

- It is an ‘ordinary’ queue, that is, it is not an initiation queue or a transmission queue, and it does not generate trigger messages.
- The maximum queue depth is 1000 messages; the maximum message length is 2000 bytes.

The following MQSC command does this:

```
DEFINE QLOCAL (ORANGE.LOCAL.QUEUE) +
       DESCR('Queue for messages from other systems') +
       PUT (DISABLED) +
       GET (ENABLED) +
       NOTRIGGER +
       MSGDLVSQ (FIFO) +
       MAXDEPTH (1000) +
       MAXMSGL (2000) +
       USAGE (NORMAL)
```

Notes:

1. Most of these attributes are the defaults as supplied with the product. However, they are shown here for purposes of illustration. You can omit them if you are sure that the defaults are what you want or have not been changed. See also “Displaying default object attributes”.
2. USAGE (NORMAL) indicates that this queue is not a transmission queue.
3. If you already have a local queue on the same queue manager with the name ORANGE.LOCAL.QUEUE, this command fails. Use the REPLACE attribute, if you want to overwrite the existing definition of a queue, but see also “Changing local queue attributes” on page 43.

Defining a dead-letter queue

Each queue manager should have a local queue to be used as a dead-letter queue so that messages that cannot be delivered to their correct destination can be stored for later retrieval. You must explicitly tell the queue manager about the dead-letter queue. You can do this by specifying a dead-letter queue on the **crtmqm** command or you can use the ALTER QMGR command to specify one later. You must also define the dead-letter queue before it can be used.

A sample dead-letter queue called SYSTEM.DEAD.LETTER.QUEUE is supplied with the product. This queue is automatically created when you run the sample. You can modify this definition, if required. There is no need to rename it.

A dead-letter queue has no special requirements except that

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

MQSeries provides a dead-letter queue handler that allows you to specify how messages found on a dead-letter queue are to be processed or removed. For further information, see “Chapter 8. The MQSeries dead-letter queue handler” on page 93.

Displaying default object attributes

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

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

Note: The syntax of this command is different from that of the corresponding DEFINE command.

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

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

This command displays the three specified attributes as follows:

```
AMQ8409: Display Queue details.
    QUEUE(ORANGE.LOCAL.QUEUE)
    MAXDEPTH(1000)
    MAXMSGL(2000)
    CURDEPTH(0)
```

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

Copying a local queue definition

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

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

This command creates a queue with the same attributes as our original queue ORANGE.LOCAL.QUEUE, rather than those of the system default local queue.

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

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

This command copies the attributes of the queue ORANGE.LOCAL.QUEUE to the queue THIRD.QUEUE, but specifies that the maximum message length on the new queue is to be 1024 bytes, rather than 2000.

Notes:

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

Changing local queue attributes

You can change queue attributes in two ways, using either the ALTER QLOCAL command or the DEFINE QLOCAL command with the REPLACE attribute. In "Defining a local queue" on page 40, we defined the queue ORANGE.LOCAL.QUEUE. Suppose, for example, you wanted to increase the maximum message length on this queue to 10 000 bytes.

- Using the ALTER command:

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

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

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

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

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

Clearing a local queue

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

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

You cannot clear a queue if:

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

Deleting a local queue

Use the MQSC command DELETE QLOCAL to delete a local queue. A queue cannot be deleted if it has uncommitted messages on it. However, if the queue has one or more committed messages, and no uncommitted messages, it can only be deleted if you specify the PURGE option. For example:

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

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

Browsing queues

If you need to look at the contents of the messages on a queue, MQSeries for OpenVMS provides a sample queue browser for this purpose. The browser is supplied both as source and as a module that can be run. By default, the file names and paths are:

Source
: `MQS_EXAMPLES:AMQSBCG0.C`

Executable
: `[.BIN]AMQSBCG.EXE`, under `MQS_EXAMPLES:`

.

The sample takes two parameters, which are the:
- Queue name, for example, `SYSTEM.ADMIN.RESPQ.TEST`.
- Queue manager name, for example, `JJJH`

as shown in the following command:

```
  amqsbcg "SYSTEM.ADMIN.RESPQ.TEST" "JJJH"
```

There are no defaults; both parameters are required. Typical results from this commands are shown in Figure 5 on page 45.

```
$ amqsbcg "SYSTEM.ADMIN.RESPQ.TEST" "JJJH"

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

 MQOPEN - 'SYSTEM.ADMIN.RESPQ.TEST'


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

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 819
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D51204A4A4A48202020202020202020206EC8753A13200000'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                 '
  ReplyToQMgr    : 'JJJH                                             '
  ** Identity Context
  UserIdentifier : 'SYSTEM      '
  AccountingToken :
   X'0536353534300000000000000000000000000000000000000000000000000006'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '12'
  PutApplName    : 'AMQSPUT.EXE                 '
  PutDate  : '20010129'    PutTime  : '19483901'
  ApplOriginData : '    '

  GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '14'

****   Message      ****

 length - 14 bytes

00000000:  7465 7374 206D 6573 7361 6765 2031      'test message 1  '
```

Figure 5. Typical results from a queue browser (Part 1 of 3)

```
 MQGET of message number 2
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 819
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D51204A4A4A4820202020202020206EC8753A23200000'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'JJJH                                            '
  ** Identity Context
  UserIdentifier : 'SYSTEM      '
  AccountingToken :
   X'0536353534300000000000000000000000000000000000000000000000000006'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '12'
  PutApplName    : 'AMQSPUT.EXE                 '
  PutDate  : '20010129'    PutTime  : '19484323'
  ApplOriginData : '    '

  GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '14'

****   Message      ****

 length - 14 bytes

00000000:  6D65 7373 6167 6520 3220 4441 5441      'message 2 DATA  '


 MQGET of message number 3
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 819
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D51204A4A4A4820202020202020206EC8753A33200000'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'JJJH                                            '
  ** Identity Context
  UserIdentifier : 'SYSTEM      '
  AccountingToken :
   X'0536353534300000000000000000000000000000000000000000000000000006'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '12'
  PutApplName    : 'AMQSPUT.EXE                 '
  PutDate  : '20010129'    PutTime  : '19491145'
  ApplOriginData : '    '
```

Figure 5. Typical results from a queue browser (Part 2 of 3)

```
GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '28'

****   Message      ****

 length - 28 bytes

00000000:  6D65 7373 6167 6520 3320 6461 7461 202D 'message 3 data -'
00000010:  2065 6E64 206F 6620 696E 666F           ' end of info    '


 MQGET of message number 4
****Message descriptor****

  StrucId  : 'MD  '  Version : 2
  Report   : 0  MsgType : 8
  Expiry   : -1  Feedback : 0
  Encoding : 546  CodedCharSetId : 819
  Format : 'MQSTR   '
  Priority : 0  Persistence : 0
  MsgId : X'414D51204A4A4A48202020202020202020206EC8753A43200000'
  CorrelId : X'000000000000000000000000000000000000000000000000'
  BackoutCount : 0
  ReplyToQ       : '                                                '
  ReplyToQMgr    : 'JJJH                                            '
  ** Identity Context
  UserIdentifier : 'SYSTEM      '
  AccountingToken :
   X'0536353534300000000000000000000000000000000000000000000000000006'
  ApplIdentityData : '                                '
  ** Origin Context
  PutApplType    : '12'
  PutApplName    : 'AMQSPUT.EXE                 '
  PutDate  : '20010129'    PutTime  : '19510318'
  ApplOriginData : '    '

  GroupId : X'000000000000000000000000000000000000000000000000'
  MsgSeqNumber   : '1'
  Offset         : '0'
  MsgFlags       : '0'
  OriginalLength : '81'

****   Message      ****

 length - 81 bytes

00000000:  4A4F 484E 534F 4E2C 4441 5649 4420 4D52 'JOHNSON,DAVID MR'
00000010:  2020 2020 3239 2D4A 414E 2D32 3030 3120 '    29-JAN-2001 '
00000020:  3133 3A34 3220 3431 3233 3030 3831 2031 '13:42 41230081 1'
00000030:  3238 332E 3334 2020 3030 3235 2E32 3220 '283.34  0025.22 '
00000040:  2030 3030 302E 3030 2020 3739 3235 2E36 ' 0000.00  7925.6'
00000050:  35                                      '5               '



 No more messages
 MQCLOSE
 MQDISC
$
```

Figure 5. Typical results from a queue browser (Part 3 of 3)

Working with alias queues

An alias queue (also known as a queue alias) provides a method of redirecting MQI calls. An alias queue is not a real queue but a definition that resolves to a real queue. The alias queue definition contains a target queue name which is specified by the TARGQ attribute (*BaseQName* in PCF). When an application specifies an alias queue in an MQI call, the queue manager resolves the real queue name at run time.

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

By changing the value of the TARGQ attribute, you can redirect MQI calls to another queue, possibly on another queue manager. This is useful for maintenance, migration, and load balancing.

Defining an alias queue

The following command creates an alias queue:

```
DEFINE QALIAS (MY.ALIAS.QUEUE) TARGQ (YELLOW.QUEUE)
```

This command redirects MQI calls that specify MY.ALIAS.QUEUE, to the queue YELLOW.QUEUE. The command does not create the target queue; the MQI calls fail if the queue YELLOW.QUEUE does not exist at run time.

If you change the alias definition, you can redirect the MQI calls to another queue. For example:

```
DEFINE QALIAS (MY.ALIAS.QUEUE) TARGQ (MAGENTA.QUEUE) REPLACE
```

This command redirects MQI calls to another queue, MAGENTA.QUEUE.

You can also use alias queues to make a single queue (the target queue) appear to have different attributes for different applications. You do this by defining two aliases, one for each application. Suppose there are two applications:

- Application ALPHA can put messages on YELLOW.QUEUE, but is not allowed to get messages from it.
- Application BETA can get messages from YELLOW.QUEUE, but is not allowed to put messages on it.

You can do this using the following commands:

```
* This alias is put enabled and get disabled for application ALPHA

DEFINE QALIAS (ALPHAS.ALIAS.QUEUE) +
       TARGQ (YELLOW.QUEUE) +
       PUT (ENABLED) +
       GET (DISABLED)

* This alias is put disabled and get enabled for application BETA

DEFINE QALIAS (BETAS.ALIAS.QUEUE) +
       TARGQ (YELLOW.QUEUE) +
       PUT (DISABLED) +
       GET (ENABLED)
```

ALPHA uses the queue name ALPHAS.ALIAS.QUEUE in its MQI calls; BETA uses the queue name BETAS.ALIAS.QUEUE. They both access the same queue, but in different ways.

You can use the LIKE and REPLACE attributes when you define queue aliases, in the same way that you use them with local queues.

Using other commands with queue aliases

You can use the appropriate MQSC commands to display or alter queue alias attributes, or delete the queue alias object. For example:

```
* Display the queue alias' attributes
* ALL = Display all attributes

DISPLAY QUEUE (ALPHAS.ALIAS.QUEUE) ALL


* ALTER the base queue name, to which the alias resolves.
* FORCE = Force the change even if the queue is open.

ALTER QALIAS (ALPHAS.ALIAS.QUEUE) TARGQ(ORANGE.LOCAL.QUEUE) FORCE


* Delete this queue alias, if you can.

DELETE QALIAS (ALPHAS.ALIAS.QUEUE)
```

You cannot delete a queue alias if, for example, an application currently has the queue open or has a queue open that resolves to this queue. See the *MQSeries Command Reference* book for more information about this and other queue alias commands.

Working with model queues

A queue manager creates a *dynamic queue* if it receives an MQI call from an application specifying a queue name that has been defined as a model queue. The name of the new dynamic queue is generated by the queue manager when the queue is created. A *model queue* is a template that specifies the attributes of any dynamic queues created from it.

Working with model queues

Model queues provide a convenient method for applications to create queues as they are required.

Defining a model queue

You define a model queue with a set of attributes in the same way that you define a local queue. Model queues and local queues have the same set of attributes except that on model queues you can specify whether the dynamic queues created are temporary or permanent. (Permanent queues are maintained across queue manager restarts, temporary ones are not.) For example:

```
DEFINE QMODEL (GREEN.MODEL.QUEUE) +
       DESCR('Queue for messages from application X') +
       PUT (DISABLED) +
       GET (ENABLED) +
       NOTRIGGER +
       MSGDLVSQ (FIFO) +
       MAXDEPTH (1000) +
       MAXMSGL (2000) +
       USAGE (NORMAL) +
       DEFTYPE (PERDYN)
```

This command creates a model queue definition. From the DEFTYPE attribute, the actual queues created from this template are permanent dynamic queues.

Note: The attributes not specified are automatically copied from the SYSYTEM.DEFAULT.MODEL.QUEUE default queue.

You can use the LIKE and REPLACE attributes when you define model queues, in the same way that you use them with local queues.

Using other commands with model queues

You can use the appropriate MQSC commands to display or alter a model queue's attributes, or delete the model queue object. For example:

```
* Display the model queue's attributes
* ALL = Display all attributes

DISPLAY QUEUE (GREEN.MODEL.QUEUE) ALL


* ALTER the model to enable puts on any
* dynamic queue created from this model.

ALTER QMODEL (BLUE.MODEL.QUEUE) PUT(ENABLED)


* Delete this model queue:

DELETE QMODEL (RED.MODEL.QUEUE)
```

Managing objects for triggering

MQSeries provides a facility for starting an application automatically when certain conditions on a queue are met. One example of the conditions is when the number of messages on a queue reaches a specified number. This facility is called *triggering* and is described in detail in the *MQSeries Application Programming Guide*. This section describes how to set up the required objects to support triggering on MQSeries for Compaq OpenVMS.

Defining an application queue for triggering

An application queue is a local queue that is used by applications for messaging, through the MQI. Triggering requires a number of queue attributes to be defined on the application queue. Triggering itself is enabled by the *Trigger* attribute (TRIGGER in MQSC).

In this example, a trigger event is to be generated when there are 100 messages of priority 5 or greater on the local queue MOTOR.INS.QUEUE, as follows:

```
DEFINE QLOCAL (MOTOR.INS.QUEUE) +
       PROCESS (MOTOR.INS.PROC) +
       MAXMSGL (2000) +
       DEFPSIST (YES) +
       INITQ (MOTOR.INS.INT.Q) +
       TRIGGER +
       TRIGTYPE (DEPTH) +
       TRIGDPTH (100)+
       TRIGMPRI (5)
```

Where:

QLOCAL (MOTOR.INS.QUEUE)
Specifies the name of the application queue being defined.

PROCESS (MOTOR.INS.PROC)
Specifies the name of the application to be started by a trigger monitor program.

MAXMSGL (2000)
Specifies the maximum length of messages on the queue.

DEFPSIST (YES)
Specifies that messages are persistent on this queue.

INITQ (MOTOR.INS.INT.Q)
Is the name of the initiation queue on which the queue manager is to put the trigger message.

TRIGGER
Is the trigger attribute value.

TRIGTYPE (DEPTH)
Specifies that a trigger event is generated when the number of messages of the required priority (TRIMPRI) reaches the number specified in TRIGDPTH.

TRIGDPTH (100)
Specifies the number of messages required to generate a trigger event.

TRIGMPRI (5)
Is the priority of messages that are to be counted by the queue manager in deciding whether to generate a trigger event. Only messages with priority 5 or higher are counted.

Defining an initiation queue

When a trigger event occurs, the queue manager puts a trigger message on the initiation queue specified in the application queue definition. Initiation queues have no special settings, but you can use the following definition of the local queue MOTOR.INS.INT.Q for guidance:

```
DEFINE QLOCAL(MOTOR.INS.INT.Q) +
       GET (ENABLED) +
       NOSHARE +
       NOTRIGGER +
       MAXMSGL (2000) +
       MAXDEPTH (10)
```

Creating a process definition

Use the DEFINE PROCESS command to create a process definition. A process definition associates an application queue with the application that is to process messages from the queue. This is done through the PROCESS attribute on the application queue MOTOR.INS.QUEUE. The following MQSC command defines the required process, MOTOR.INS.PROC, identified in this example:

```
DEFINE PROCESS (MOTOR.INS.PROC) +
               DESCR ('Insurance request message processing') +
               APPLTYPE (OPENVMS) +
               APPLICID ('DKA0:[MQM.ADMIN.TEST]IRMP01.EXE') +
               USERDATA ('open, close, 235')
```

Where:

MOTOR.INS.PROC
Is the name of the process definition, limited to 15 characters.

DESCR ('Insurance request message processing')
Is the descriptive text of the application program to which the definition relates, following the keyword. This text is displayed when you use the DISPLAY PROCESS command. This can help you to identify what the process does. If you use spaces in the string, you must enclose the string in single quotes.

APPLTYPE (OPENVMS)
Is the type of the application that runs on OpenVMS.

APPLICID ('DKA0:[MQM.ADMIN.TEST]IRMP01.EXE')
Is the name of the application executable program.

USERDATA ('open, close, 235')
Is user-defined data, which can be used by the application.

Displaying your process definition

Use the DISPLAY PROCESS command, with the ALL keyword, to examine the results of your definition. For example:

```
DISPLAY PROCESS (MOTOR.INS.PROC) ALL


    24 : DISPLAY PROCESS (MOTOR.INS.PROC) ALL
AMQ8407: Display Process details.
    DESCR ('Insurance request message processing') +
    APPLICID ('DKA0:[MQM.ADMIN.TEST]IRMP01.EXE') +
    USERDATA (open, close, 235) +
    PROCESS (MOTOR.INS.PROC) +
    APPLTYPE (OPENVMS)
```

You can also use the MQSC command ALTER PROCESS to alter an existing process definition and DELETE PROCESS to delete a process definition.

Chapter 5. Automating administration tasks

This chapter assumes that you have experience of administering MQSeries objects.

There may come a time when you decide that it would be beneficial to your installation to automate some administration and monitoring tasks. You can automate administration tasks for both local and remote queue managers using programmable command format (PCF) commands.

This chapter describes:
- How to use programmable command formats to automate administration tasks in "PCF commands"
- How to use the command server in "Managing the command server for remote administration" on page 57

PCF commands

The purpose of MQSeries programmable command format (PCF) commands is to allow administration tasks to be programmed into an administration program. In this way you can create queues, process definitions, channels, and namelists, and change queue managers, from a program.

PCF commands cover the same range of functions provided by the MQSC facility.

Therefore, you can write a program to issue PCF commands to any queue manager in the network from a single node. In this way, you can both centralize and automate administration tasks.

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

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

Briefly, these are some of the things the application programmer must specify to create a PCF command message:

Message descriptor
This is a standard MQSeries message descriptor, in which:
Message type (*MsgType*) is MQMT_REQUEST.
Message format (*Format*) is MQFMT_ADMIN.

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

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

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

For a complete description of the PCF data structures and how to implement them, see the *MQSeries Programmable System Management* book.

Attributes in MQSC and PCFs

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

Object attributes in PCF, which are not limited to eight characters, are shown in this book in italics. For example, the PCF equivalent of RQMNAME is *RemoteQMgrName*.

Escape PCFs

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

Using the MQAI to simplify the use of PCFs

The MQAI is an administration interface to MQSeries that is available on the OpenVMS platform.

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

The MQAI can be used:

- **To simplify the use of PCF messages**

  The MQAI is an easy way to administer MQSeries; you do not have to write your own PCF messages and this avoids the problems associated with complex data structures.

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

  On the other hand, programs written using the MQAI pass parameters into the appropriate data bag and only one statement is required for each structure. The use of MQAI data bags removes the need for you to handle arrays and allocate storage, and provides some degree of isolation from the details of the PCF.
- **To implement self-administering applications and administration tools**

  For example, the Active Directory Services provided by MQSeries for Windows NT and Windows 2000 Version 5.2 uses the MQAI. (There is currently no example of this usage on OpenVMS platform.)
- **To handle error conditions more easily**

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

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

For more information about using the MQAI, see the *MQSeries Administration Interface Programming Guide and Reference* book.

For more information about PCFs in general, see the *MQSeries Programmable System Management* book.

Managing the command server for remote administration

Each queue manager can have a command server associated with it. A command server processes any incoming commands from remote queue managers, or PCF commands from applications. It presents the commands to the queue manager for processing and returns a completion code or operator message depending on the origin of the command.

A command server is mandatory for all administration involving PCFs, the MQAI, and also for remote administration.

Note: For remote administration, you must ensure that the target queue manager is running. Otherwise, the messages containing commands cannot leave the queue manager from which they are issued. Instead, these messages are queued in the local transmission queue that serves the remote queue manager. This situation should be avoided, if at all possible.

Starting the command server

To start the command server use this command:

```
strmqcsv "saturn.queue.manager"
```

where `saturn.queue.manager` is the queue manager for which the command server is being started.

Displaying the status of the command server

For remote administration, ensure that the command server on the target queue manager is running. If it is not running, remote commands cannot be processed. Any messages containing commands are queued in the target queue manager’s command queue.

To display the status of the command server for a queue manager, called here `saturn.queue.manager`, the command is:

```
dspmqcsv "saturn.queue.manager"
```

You must issue this command on the target machine. If the command server is running, the following message is returned:

```
AMQ8027    MQSeries Command Server Status ..: Running
```

Stopping a command server

To end a command server, the command, using the previous example is:

```
endmqcsv "saturn.queue.manager"
```

You can stop the command server in two different ways:

- For a controlled stop, use the **endmqcsv** command with the -c flag, which is the default.
- For an immediate stop, use the **endmqcsv** command with the -i flag.

Note: Stopping a queue manager also ends the command server associated with it (if one has been started).

Chapter 6. Administering remote MQSeries objects

This chapter describes how to administer MQSeries objects on another queue manager. It also describes how you can use remote queue objects to control the destination of messages and reply messages.

It contains these sections:

For more information about channels, their attributes, and how to set them up, refer to the *MQSeries Intercommunication* book.

Channels, clusters and remote queuing

A queue manager communicates with another queue manager by sending a message and, if required, receiving back a response. The receiving queue manager could be:

- On the same machine
- On another machine in the same location or on the other side of the world
- Running on the same platform as the local queue manager
- Running on another platform supported by MQSeries

These messages may originate from:

- User-written application programs that transfer data from one node to another.
- User-written administration applications that use PCFs, the MQAI, or the ADSI.
- Queue managers sending:
  - Instrumentation event messages to another queue manager.
  - MQSC commands issued from a **runmqsc** command in indirect mode (where the commands are run on another queue manager).

Before a message can be sent to a remote queue manager, the local queue manager needs a mechanism to detect the arrival of messages and transport them consisting:

- Of at least one channel
- A transmission queue
- A message channel agent (MCA)
- A channel listener
- A channel initiator

A channel is a one-way communication link between two queue managers and can carry messages destined for any number of queues at the remote queue manager.

Each end of the channel has a separate definition. For example, if one end is a sender or a server, the other end must be a receiver or a requester. A simple channel consists of a *sender channel definition* at the local queue manager end and a *receiver channel definition* at the remote queue manager end. The two definitions must have the same name and together constitute a single channel.

Administering remote objects

If the remote queue manager is expected to respond to messages sent by the local queue manager, a second channel needs to be set up to send responses back to the local queue manager.

Channels are defined using the MQSC DEFINE CHANNEL command. In this chapter, the examples relating to channels use the default channel attributes unless otherwise specified.

There is a message channel agent (MCA) at each end of a channel which controls the sending and receiving of messages. It is the job of the MCA to take messages from the transmission queue and put them on the communication link between the queue managers.

A transmission queue is a specialized local queue that temporarily holds messages before they are picked up by the MCA and sent to the remote queue manager. You specify the name of the transmission queue on a *remote queue definition*.

"Preparing channels and transmission queues for remote administration" on page 62 shows how to use these definitions to set up remote administration.

For more information about setting up distributed queuing in general, see the *MQSeries Intercommunication* book.

Remote administration using clusters

In a traditional MQSeries network using distributed queuing, every queue manager is independent. If one queue manager needs to send messages to another queue manager it must have defined a transmission queue, a channel to the remote queue manager, and a remote queue definition for every queue to which it wants to send messages.

A *cluster* is a group of queue managers set up in such a way so that the queue managers can communicate directly with one another over a single network without the need for complex transmission queue, channel, and queue definitions. Clusters can be set up easily, and typically contain queue managers that are logically related in some way and need to share data or applications.

Once a cluster has been created the queue managers within it can communicate with each other *without the need for complicated channel or remote queue definitions*. Even the smallest cluster will reduce system administration overheads.

Establishing a network of queue managers in a cluster involves fewer definitions than establishing a traditional distributed queuing environment. With fewer definitions to make, you can set up or change your network more quickly and easily, and the risk in making an error in your definitions is reduced.

To set up a cluster, you usually need one cluster sender (CLUSSDR) definition and one cluster receiver (CLUSRCVR) definition per queue manager. You do not need any transmission queue definitions or remote queue definitions. The principles of remote administration are the same when used within a cluster, but the definitions themselves are greatly simplified.

For more information about clusters, their attributes, and how to set them up, refer to the *MQSeries Queue Manager Clusters* book.

Remote administration from a local queue manager using MQSC commands

This section tells you how to administer a remote queue manager from a local queue manager. You can implement remote administration from a local node using:
- MQSC commands
- PCF commands

Preparing the queues and channels is essentially the same for both methods. In this book, the examples show MQSC commands, because they are easier to understand. However, you can convert the examples to PCFs if you wish. For more information about writing administration programs using PCFs, see the *MQSeries Programmable System Management* book.

In remote administration you send MQSC commands to a remote queue manager—either interactively or from a text file containing the commands. The remote queue manager may be on the same machine or, more typically, on a different machine. You can remotely administer queue managers in different MQSeries environments, including AIX®, AS/400, MVS/ESA, and OS/2®.

To implement remote administration, you must create certain objects. Unless you have specialized requirements, you should find that the default values (for example, for message length) are sufficient.

Preparing queue managers for remote administration

Figure 6 on page 62 shows the configuration of queue managers and channels that are required for remote administration using the **runmqsc** command. `source.queue.manager` is the *source* queue manager from which you can issue MQSC commands and to which the results of these commands (operator messages) are returned, if possible. `target.queue.manager` is the destination queue manager, which processes the commands and generates any operator messages.

Note: `source.queue.manager` ***must*** be the default queue manager. For further information on creating a queue manager, see “crtmqm (Create queue manager)” on page 231.

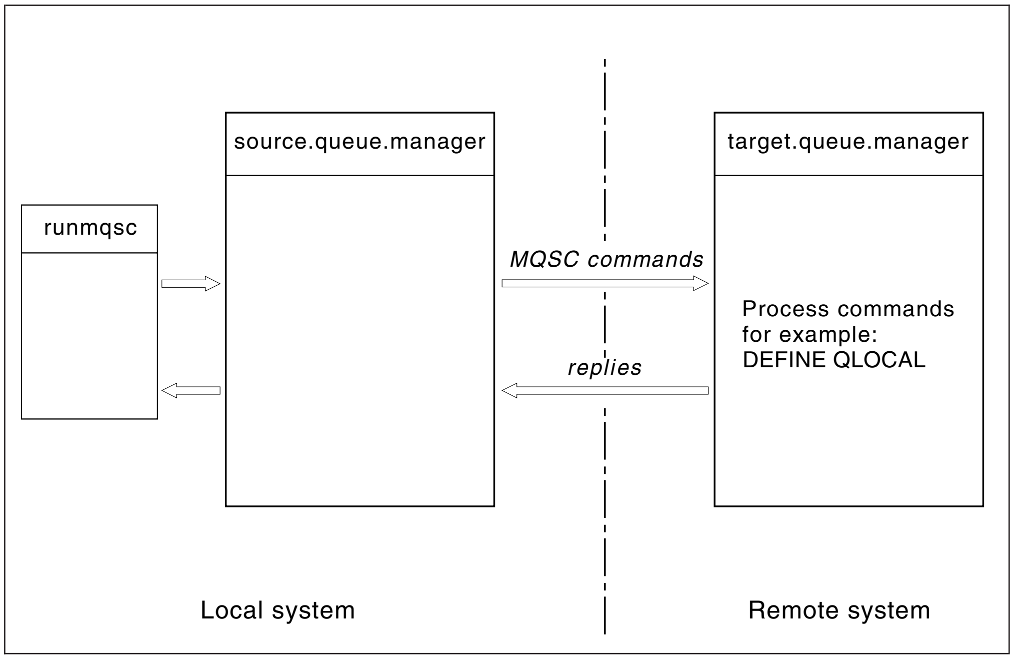


Figure 6. Remote administration

On both systems, if you have not already done so, you must:
- Create the queue manager, using the **crtmqm** command.
- Start the queue manager, using the **strmqm** command.

You have to run these commands locally or over a network facility, for example Telnet.

On the destination queue manager:
- The command queue, SYSTEM.ADMIN.COMMAND.QUEUE, must be present. This queue is created by default when a queue manager is created.
- The command server must be started, using the **strmqcsv** command.

Preparing channels and transmission queues for remote administration

To run MQSC commands remotely, you must set up two channels, one for each direction, and their associated transmission queues. This example assumes that TCP/IP is being used as the transport type and that you know the TCP/IP address involved.

The channel `source.to.target` is for sending MQSC commands from the source queue manager to the destination. Its sender is at `source.queue.manager` and its receiver is at queue manager `target.queue.manager`. The channel `target.to.source` is for returning the output from commands and any operator messages that are generated to the source queue manager. You must also define a transmission queue for each sender. This queue is a local queue that is given the name of the receiving queue manager. The XMITQ name must match the remote queue manager name for remote administration to work, unless you are using a queue manager alias. Figure 7 on page 63 summarizes this configuration.

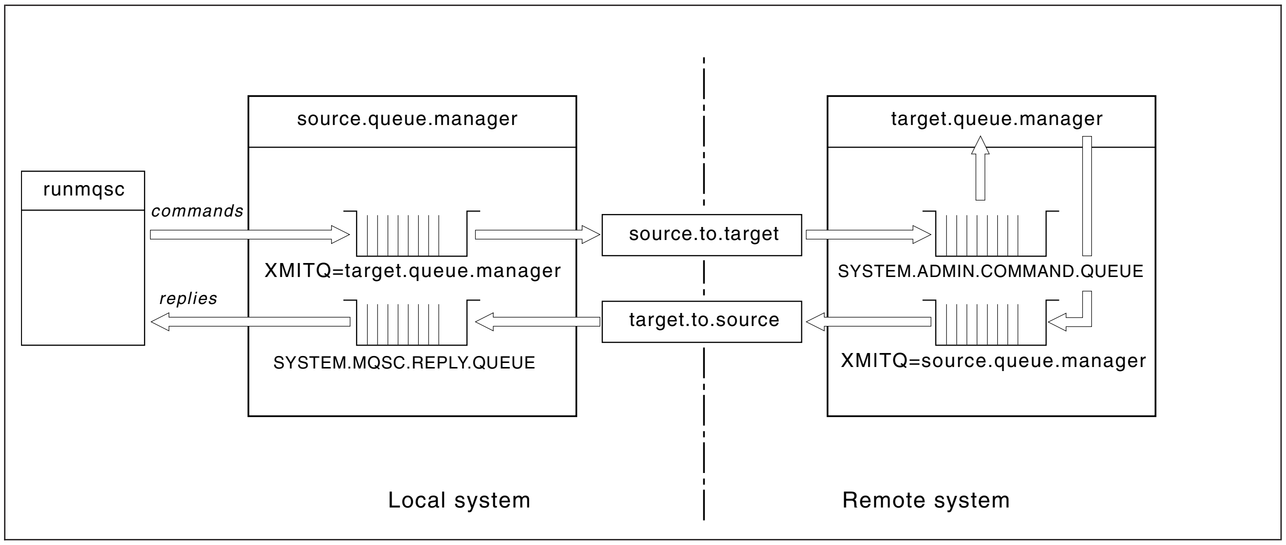


Figure 7. Setting up channels and queues for remote administration

See the *MQSeries Intercommunication* book for more information about setting up remote channels.

Defining channels and transmission queues

On the source queue manager, issue these MQSC commands to define the channels and the transmission queue:

```
* Define the sender channel at the source queue manager

DEFINE CHANNEL ('source.to.target') +
       CHLTYPE(SDR) +
       CONNAME (RHX5498) +
       XMITQ ('target.queue.manager') +
       TRPTYPE(TCP)

* Define the receiver channel at the source queue manager

DEFINE CHANNEL ('target.to.source') +
       CHLTYPE(RCVR) +
       TRPTYPE(TCP)

* Define the transmission queue on the source

DEFINE QLOCAL ('target.queue.manager') +
       USAGE (XMITQ)
```

Remote administration

Issue these commands on the destination queue manager (`target.queue.manager`), to create the channels and the transmission queue there:

```
* Define the sender channel on the destination queue manager

DEFINE CHANNEL ('target.to.source') +
       CHLTYPE(SDR) +
       CONNAME (RHX7721) +
       XMITQ ('source.queue.manager') +
       TRPTYPE(TCP)

* Define the receiver channel on the destination queue manager

DEFINE CHANNEL ('source.to.target') +
       CHLTYPE(RCVR) +
       TRPTYPE(TCP)

* Define the transmission queue on the destination queue manager

DEFINE QLOCAL ('source.queue.manager') +
       USAGE (XMITQ)
```

Note: The TCP/IP connection names specified for the CONNAME attribute in the sender channel definitions are for illustration only. This is the network name of the machine at the *other* end of the connection. Use the values appropriate for your network.

Starting the channels

The following description assumes that both ends of the channel are running on MQSeries for Compaq OpenVMS. If this is not the case, refer to the relevant documentation for the non-OpenVMS end of the channel.

To start the two channels, first ensure that the TCP/IP services have been configured for MQSeries on both nodes, and are running at both ends of the connections.

Start a listener at the receiver end of each channel.

- On the source queue manager, type:

```
runmqlsr -m "source.queue.manager" -t tcp
```

- On the destination queue manager, type:

```
runmqlsr -m "target.queue.manager" -t tcp
```

The queue manager name is not required on the **runmqchl** command for the source queue manager because the source queue manager must be the default queue manager. The queue manager name is required on the **runmchl** command for the destination queue manager if the destination queue manager is not the default queue manager on its node.

Then start the channels:

- On the source queue manager, type:

```
runmqchl -m "source.queue.manager" -c "source.to.target"
```

- On the destination queue manager, type:

```
runmqchl -m "target.queue.manager" -c "target.to.source"
```

The **runmqlsr** and **runmqchl** commands are MQSeries control commands. They cannot be issued using **runmqsc**. Listeners and channels can be started using **runmqsc** commands or scripts (start channel and start listener).

Automatic definition of channels

Automatic definition of channels applies only if the destination queue manager is running on MQSeries Version 5.1, or later, products. If an inbound attach request is received and an appropriate receiver or server-connection definition cannot be found in the channel definition file (CDF), MQSeries creates a definition automatically and adds it to the CDF. Automatic definitions are based on two default definitions supplied with MQSeries: SYSTEM.AUTO.RECEIVER and SYSTEM.AUTO.SVRCONN.

You enable automatic definition of receiver and server-connection definitions by updating the queue manager object using the MQSC command, ALTER QMGR (or the PCF command Change Queue Manager).

For more information about the automatic creation of channel definitions, see the *MQSeries Intercommunication* book.

For information about the automatic definition of channels for clusters, see the *MQSeries Queue Manager Clusters* book.

Issuing MQSC commands remotely

The command server ***must*** be running on the destination queue manager, if it is going to process MQSC commands remotely. (This is not necessary on the source queue manager.)

- On the destination queue manager, type:

```
strmqcsv "target.queue.manager"
```

- On the source queue manager, you can then run MQSC interactively in queued mode by typing:

```
runmqsc -w 30 "target.queue.manager"
```

This form of the **runmqsc** command—with the -w flag—runs the MQSC commands in queued mode, where commands are put (in a modified form) on the command-server input queue and executed in order.

Remote administration

When you type in an MQSC command, it is redirected to the remote queue manager, in this case, `target.queue.manager`. The timeout is set to 30 seconds; if a reply is not received within 30 seconds, the following message is generated on the local (source) queue manager:

```
AMQ8416: MQSC timed out waiting for a response from the command server.
```

At the end of the MQSC session, the local queue manager displays any timed-out responses that have arrived. When the MQSC session is finished, any further responses are discarded.

In indirect mode, you can also run an MQSC command file on a remote queue manager. For example:

```
runmqsc -w 60 "target.queue.manager" < mycomds.in > report.out
```

where `mycomds.in` is a file containing MQSC commands and `report.out` is the report file.

Working with queue managers on MVS/ESA

You can issue MQSC commands to an MVS/ESA queue manager from an MQSeries for Compaq OpenVMS queue manager. However, to do this, you must modify the **runmqsc** command and the channel definitions at the sender.

In particular, you add the -x flag to the **runmqsc** command on an OpenVMS node:

```
runmqsc -w 30 -x "target.queue.manager"
```

On the sender channel, set the CONVERT attribute to YES. This specifies that the required data conversion between the systems is performed at the OpenVMS end. The channel definition command now becomes:

```
* Define the sender channel at the source queue manager on OpenVMS

DEFINE CHANNEL ('source.to.target') +
       CHLTYPE(SDR) +
       CONNAME (RHX5498) +
       XMITQ ('target.queue.manager') +
       TRPTYPE(TCP) +
       CONVERT (YES)
```

You must also define the receiver channel and the transmission queue at the source queue manager as before. Again, this example assumes that TCP/IP is the transmission protocol being used.

Recommendations for remote queuing

When you are implementing remote queuing:

1. Put the MQSC commands to be run on the remote system in a command file.
2. Verify your MQSC commands locally, by specifying the -v flag on the **runmqsc** command.

You cannot use **runmqsc** to verify MQSC commands on another queue manager.

3. Check, as far as possible, that the command file runs locally without error.
4. Finally, run the command file against the remote system.

If you have problems using MQSC remotely

If you have difficulty in running MQSC commands remotely, use the following checklist to see if you have:

- Started the command server on the destination queue manager.
- Defined a valid transmission queue.
- Defined the two ends of the message channels for both:
  - The channel along which the commands are being sent.
  - The channel along which the replies are to be returned.
- Specified the correct connection name (CONNAME) in the channel definition.
- Started the listeners before you started the message channels.
- Checked that the disconnect interval has not expired, for example, if a channel started but then shut down after some time. This is especially important if you start the channels manually.
- Ensured that you are not sending requests from a source queue manager that do not make sense to the destination queue manager (for example, requests that include new parameters).

See also "Resolving problems with MQSC" on page 39.

Creating a local definition of a remote queue

You can use a remote queue definition as a local definition of a remote queue. You create a remote queue object on your local queue manager to identify a local queue on another queue manager.

Understanding how local definitions of remote queues work

An application connects to a local queue manager and then issues an MQOPEN call. In the open call, the queue name specified is that of a remote queue definition on the local queue manager. The remote queue definition supplies the names of the destination queue, the destination queue manager, and optionally, a transmission queue. To put a message on the remote queue, the application issues an MQPUT call, specifying the handle returned from the MQOPEN call. The queue manager appends the remote queue name and the remote queue manager name to a transmission header in the message. This information is used to route the message to its correct destination in the network.

As administrator, you can control the destination of the message by altering the remote queue definition.

Example

Purpose: An application is required to put a message on a queue owned by a remote queue manager.

How it works: The application connects to a queue manager, for example, `saturn.queue.manager`. The destination queue is owned by another queue manager.

Local definition of remote queue

On the MQOPEN call, the application specifies these fields:

Field value	Description
ObjectName CYAN.REMOTE.QUEUE	Specifies the local name of the remote queue object. This defines the destination queue and the destination queue manager.
ObjectType (Queue)	Identifies this object as a queue.
ObjectQmgrName Blank or `saturn.queue.manager`	This field is optional. If blank, the name of the local queue manager is assumed. (This is the queue manager on which the remote queue definition was made and to which the application is connected.) If not blank, the name of the local queue manager must be specified.

After this, the application issues an MQPUT call to put a message on to this queue.

On the local queue manager, you can create a local definition of a remote queue using the following MQSC commands:

```
DEFINE QREMOTE (CYAN.REMOTE.QUEUE) +
       DESCR ('Queue for auto insurance requests from the branches') +
       RNAME (AUTOMOBILE.INSURANCE.QUOTE.QUEUE) +
       RQMNAME ('jupiter.queue.manager') +
       XMITQ (INQUOTE.XMIT.QUEUE)
```

Where:

QREMOTE (CYAN.REMOTE.QUEUE)
Specifies the local name of the remote queue object. This is the name that applications connected to this queue manager must specify in the MQOPEN call to open the queue AUTOMOBILE.INSURANCE.QUOTE.QUEUE on the remote queue manager jupiter.queue.manager.

DESCR ('Queue for auto insurance requests from the branches')
Additional text that describes the use of the queue.

RNAME (AUTOMOBILE.INSURANCE.QUOTE.QUEUE)
Specifies the name of the destination queue on the remote queue manager. This is the real destination queue for messages that are sent by applications that specify the queue name CYAN.REMOTE.QUEUE. The queue AUTOMOBILE.INSURANCE.QUOTE.QUEUE must be defined as a local queue on the remote queue manager.

RQMNAME ('jupiter.queue.manager')
Specifies the name of the remote queue manager that owns the destination queue AUTOMOBILE.INSURANCE.QUOTE.QUEUE. This name must be enclosed in **single** quotations.

XMITQ (INQUOTE.XMIT.QUEUE)
Specifies the name of the transmission queue. This is optional; if not specified, a queue with the same name as the remote queue manager is used.

In either case, the appropriate transmission queue must be defined as a local queue with a *Usage* attribute specifying that it is a transmission queue (USAGE(XMIT) in MQSC).

An alternative way of putting messages on a remote queue

Using a local definition of a remote queue is not the only way of putting messages on a remote queue. Applications can specify the full queue name, which includes the remote queue manager name, as part of the MQOPEN call. In this case, a local definition of a remote queue is not required. However, this alternative means that applications must either know or have access to the name of the remote queue manager at run time.

Using other commands with remote queues

You can use the appropriate MQSC commands to display or alter the attributes of a remote queue object, or you can delete the remote queue object. For example:

```
* Display the remote queue's attributes.
* ALL = Display all attributes

DISPLAY QUEUE (CYAN.REMOTE.QUEUE) ALL


* ALTER the remote queue to enable puts.
* This does not affect the destination queue,
* only applications that specify this remote queue.

ALTER QREMOTE (CYAN.REMOTE.QUEUE) PUT(ENABLED)


* Delete this remote queue
* This does not affect the destination queue
* only its local definition

DELETE QREMOTE (CYAN.REMOTE.QUEUE)
```

Note: If you delete a remote queue, you only delete the local representation of the remote queue. You do not delete the remote queue itself or any messages on it.

Creating a transmission queue

A transmission queue is a local queue that is used when a queue manager forwards messages to a remote queue manager through a message channel. The channel provides a one-way link to the remote queue manager. Messages are queued at the transmission queue until the channel can accept them. When you define a channel, you must specify a transmission queue name at the sending end of the message channel.

The *Usage* attribute (USAGE in MQSC) defines whether a queue is a transmission queue or a normal queue.

Default transmission queues

Optionally, you can specify a transmission queue in a remote queue object, using the *XmitQName* attribute (XMITQ in MQSC). If no transmission queue is defined, a default is used. When applications put messages on a remote queue, if a transmission queue with the same name as the destination queue manager exists, that queue is used. If this queue does not exist, the queue specified by the *DefaultXmitQ* attribute (DEFXMITQ in MQSC) on the local queue manager is used.

For example, the following MQSC command creates a default transmission queue on source.queue.manager for messages going to target.queue.manager:

```
DEFINE QLOCAL ('target.queue.manager') +
       DESCR ('Default transmission queue for target qm') +
       USAGE (XMITQ)
```

Applications can put messages directly on a transmission queue, with an appropriate header, or they can be put there indirectly, for example, through a remote queue definition. See also "Creating a local definition of a remote queue" on page 67.

Using remote queue definitions as aliases

In addition to locating a queue on another queue manager, you can also use a local definition of a remote queue for both:
- Queue manager aliases
- Reply-to queue aliases

Both types of aliases are resolved through the local definition of a remote queue.

As usual in remote queuing, the appropriate channels must be set up if the message is to arrive at its destination.

Queue manager aliases

An alias is the process by which the name of the destination queue manager—as specified in a message—is modified by a queue manager on the message route. Queue manager aliases are important because you can use them to control the destination of messages within a network of queue managers.

You do this by altering the remote queue definition on the queue manager at the point of control. The sending application is not aware that the queue manager name specified is an alias.

For more information about queue manager aliases, see the *MQSeries Intercommunication* book.

Reply-to queue aliases

Optionally, an application can specify the name of a reply-to queue when it puts a *request message* on a queue. If the application that processes the message extracts the name of the reply-to queue, it knows where to send the *reply message*, if required.

A reply-to queue alias is the process by which a reply-to queue – as specified in a request message – is altered by a queue manager on the message route. The sending application is not aware that the reply-to queue name specified is an alias.

A reply-to queue alias lets you alter the name of the reply-to queue and optionally its queue manager. This in turn lets you control which route is used for reply messages.

For more information about request messages, reply messages, and reply-to queues, see the *MQSeries Application Programming Reference* book. For more information about reply-to queue aliases, see the *MQSeries Intercommunication* book.

Data conversion

Message data in MQSeries-defined formats (also known as built-in formats) can be converted by the queue manager from one coded character set to another, provided that both character sets relate to a single language or a group of similar languages.

For example, conversion between coded character sets whose identifiers (CCSIDs) are 850 and 500 is supported, because both apply to Western European languages.

For EBCDIC new line (NL) character conversions to ASCII, see “The AllQueueManagers stanza” on page 161.

Supported conversions are defined in Code page conversion tables in the *MQSeries Application Programming Reference* book.

When a queue manager cannot convert messages in built-in formats

The queue manager cannot automatically convert messages in built-in formats if their CCSIDs represent different national-language groups. For example, conversion between CCSID 850 and CCSID 1025 (which is an EBCDIC coded character set for languages using Cyrillic script) is not supported because many of the characters in one coded character set cannot be represented in the other. If you have a network of queue managers working in different national languages, and data conversion among some of the coded character sets is not supported, you can enable a default conversion. Default data conversion is described in “Default data conversion”.

File ccsid.tbl

The file ccsid.tbl specifies:

- Any additional code sets. To specify additional code sets, you need to edit ccsid.tbl (guidance on how to do this is provided in the file).
- any default data conversion.

You can update the information recorded in ccsid.tbl. You might want to do this if, for example, a future release of your operating system supports additional coded character sets.

In MQSeries for Compaq OpenVMS, a sample ccsid.tbl file is provided as

```
MQS_EXAMPLES:CCSID.TBL
```

and the active ccsid.tbl file is located in directory

```
MQS_ROOT:[MQM.CONV.TABLE]
```

Default data conversion: To implement default data conversion, you edit ccsid.tbl to specify a default EBCDIC CCSID and a default ASCII CCSID, and also to specify the defaulting CCSIDs. Instructions for doing this are included in the file.

If you update ccsid.tbl to implement default data conversion, the queue manager must be restarted before the change can take effect.

The default data conversion process is as follows:

- If conversion between the source and target CCSIDs is not supported, but the CCSIDs of the source and target environments are either both EBCDIC or both ASCII, the character data is passed to the target application without conversion.
- If one CCSID represents an ASCII coded character set, and the other represents an EBCDIC coded character set, MQSeries converts the data using the default data-conversion CCSIDs defined in ccsid.tbl.

Note: You should try to restrict the characters being converted to those that have the same code values in the coded character set specified for the message and in the default coded character set. If you use only that set of characters that is valid for MQSeries object names you will, in general, satisfy this requirement. Exceptions occur with EBCDIC CCSIDs 290, 930, 1279, and 5026 used in Japan, where the lowercase characters have different codes from those used in other EBCDIC CCSIDs.

Conversion of messages in user-defined formats

Messages in user-defined formats cannot be converted from one coded character set to another by the queue manager. If data in a user-defined format requires conversion, you must supply a data-conversion exit for each such format. The use of default CCSIDs for converting character data in user-defined formats is not recommended, although it is possible. For more information about converting data in user-defined formats and about writing data conversion exits, see the *MQSeries Application Programming Guide*.

Changing the queue manager CCSID

You are recommended to stop and restart the queue manager when you change the CCSID of the queue manager, by using the CCSID attribute of the ALTER QMGR command.

This ensures that all running applications, including the command server and channel programs, are stopped and restarted.

This is necessary, because any applications that are running when the queue manager CCSID is changed, continue to use the existing CCSID.

Chapter 7. Protecting MQSeries objects

This chapter describes the features of security control in MQSeries for Compaq OpenVMS and how you can implement this control.

It contains these sections:

Why you need to protect MQSeries resources

Because MQSeries queue managers handle the transfer of information that is potentially valuable, you need the safeguard of an authority system. This ensures that the resources that a queue manager owns and manages are protected from unauthorized access, which could lead to the loss or disclosure of the information. In a secure system, it is essential that none of the following are accessed or changed by any unauthorized user or application:

- Connections to a queue manager.
- Access to MQSeries objects such as queues, clusters, channels, and processes.
- Commands for queue manager administration, including MQSC commands and PCF commands.
- Access to MQSeries messages.
- Context information associated with messages.

You should develop your own policy with respect to which users have access to which resources.

Before you begin

All queue manager resources run with the VMS Rights Identifier:

```
MQM
```

This rights identifier is created during MQSeries installation and you **must** grant this resource attribute to all users who need to control MQSeries resources.

User IDs in MQSeries for Compaq OpenVMS with resource identifier MQM

If your user ID holds the MQM OpenVMS rights identifier, you have all authorities to all MQSeries resources. Your user ID ***must*** hold the OpenVMS MQM rights identifier to be able to use all the MQSeries for Compaq OpenVMS control commands except **crtmqcvx**. In particular, you need this authority to:

- Use the **runmqsc** command to run MQSC commands.
- Administer authorities on MQSeries for Compaq OpenVMS using the **setmqaut** command.

Before you begin

If you are sending channel commands to queue managers on a remote system, you must ensure that your user ID holds the OpenVMS rights identifier MQM on the target system. For a list of PCF and MQSC channel commands, see "Channel command security" on page 81.

In addition, installation of MQSeries creates an identifier MQS_SERVER. This is granted ownership of the resource domain where VMS keeps lock information for MQSeries. By default, access authorities to this identifier are granted to users who:
- Are in the same user group as the user MQM, or
- Are system users, or
- Have SYSPRV, SYSLCK, or BYPASS privilege set

To allow other users access to the MQ resources, you need to ensure that the MQS_SERVER identifier has appropriate WORLD privilege by running the command:

```
SET SECURITY/CLASS=RESOURCE [MQS_SERVER] /PROTECTION=(W:RWL)
```

Note: It is not essential for your user ID to hold the rights identifier MQM for issuing:
- PCF commands—including Escape PCFs—from an administration program
- MQI calls from an application program

For more information

For more information about:
- MQSeries for Compaq OpenVMS command sets, see "Chapter 2. An introduction to MQSeries administration" on page 17.
- MQSeries for Compaq OpenVMS control commands, see "Chapter 17. MQSeries control commands" on page 225.
- PCF commands and Escape PCFs, see the *MQSeries Programmable System Management* book.
- MQI calls, see the *MQSeries Application Programming Guide* and *MQSeries Application Programming Reference* book.

Understanding the Object Authority Manager

By default, access to queue manager resources is controlled through an authorization service installable component. This component is formally called the Object Authority Manager (OAM) for MQSeries for Compaq OpenVMS. It is supplied with MQSeries for Compaq OpenVMS and is automatically installed and enabled for each queue manager you create, unless you specify otherwise. In this chapter, the term OAM is used to denote the Object Authority Manager supplied with this product.

The OAM is an *installable component* of the authorization service. Providing the OAM as an installable service gives you the flexibility to:
- Replace the supplied OAM with your own authorization service component using the interface provided.
- Augment the facilities supplied by the OAM with those of your own authorization service component, again using the interface provided.
- Remove or disable the OAM and run with no authorization service at all.

For more information on installable services, see the *MQSeries Programmable System Management* book.

The OAM manages users' authorizations to manipulate MQSeries objects, including queues, process definitions, and channels. It also provides a command interface through which you can grant or revoke access authority to an object for a specific group of users. The decision to allow access to a resource is made by the OAM, and the queue manager follows that decision. If the OAM cannot make a decision, the queue manager prevents access to that resource.

How the OAM works

The OAM works by exploiting the security features of the underlying OpenVMS operating system. In particular, the OAM uses OpenVMS user, group IDs, and rights identifiers. Users can access queue manager objects only if they have the required authority.

Managing access through rights identifiers

In the command interface, we use the term *principal* rather than user ID. The reason for this is that authorities granted to a user ID can also be granted to other entities, for example, an application program that issues MQI calls, or an administration program that issues PCF commands. In these cases, the principal associated with the program is not necessarily the user ID that was used when the program was started. However, in this discussion, principals and user IDs are always OpenVMS user IDs.

Rights identifiers and the primary rights identifier

Managing access permissions to MQSeries resources is based on OpenVMS *rights identifiers*, that is, identifiers held by principals. A principal can hold one or more OpenVMS rights identifiers. A group is defined as the set of all principals that have been granted a specific rights identifier.

The OAM maintains authorizations at the level of rights identifiers rather than individual principals. The mapping of principals to identifier names is carried out within the OAM and operations are carried out at the rights identifier level. You can, however, display the authorizations of an individual principal.

When a principal holds more than one rights identifier

The authorizations that a principal has are the union of the authorizations of all the rights identifiers that it holds, that is, its process rights. Whenever a principal requests access to a resource, the OAM computes this union, and then checks the authorization against it. You can use the control command **setmqaut** to set the authorizations for a specific principal, or identifier.

Note: Any changes made using the **setmqaut** command take immediate effect, unless the object is in use. In this case, the change comes into force when the object is next opened. However, changes to a principal's rights identifier list do not come into effect until a queue manager is reset, that is, stopped and restarted.

The authorizations associated with a principal are cached when they are computed by the OAM. Any changes made to an identifier's authorizations after it has been cached are not recognized until the queue manager is restarted. Avoid changing any authorizations while the queue manager is running.

Default rights identifier

The OAM recognizes a default to which all users are nominally assigned. This group is defined by the pseudo rights identifier of 'NOBODY'. 'NOBODY' can be used as if it were a valid rights identifier to assign authorizations using MQSeries commands. By default, no authorizations are given to this identifier. Users without specific authorizations can be granted access to MQSeries resources through this rights identifier.

Resources you can protect with the OAM

Through OAM you can control:

- Access to MQSeries objects through the MQI. When an application program attempts to access an object, the OAM checks if the user ID making the request has the authorization (through the identifier held) for the operation requested.

  In particular, this means that queues, and the messages on queues, can be protected from unauthorized access.
- Permission to use MQSC commands; only principals which hold rights identifier MQM can execute queue manager administration commands, for example, to create a queue.
- Permission to use control commands; only principals which hold rights identifier MQM can execute control commands, for example, creating a queue manager, starting a command server, or using **runmqsc**.
- Permission to use PCF commands.

Different users may be granted different kinds of access authority to the same object. For example, for a specific queue, users holding one identifier may be allowed to perform both put and get operations; users with another identifier may only be allowed to browse the queue (MQGET with browse option). Similarly, users with identifiers may have get and put authority to a queue, but are not allowed to alter or delete the queue.

Using rights identifiers for authorizations

Using identifiers, rather than individual principals, for authorization reduces the amount of administration required. Typically, a particular kind of access is required by more than one principal. For example, you might define an identifier consisting of end users who want to run a particular application. New users can be given access simply by granting the appropriate identifier to their OpenVMS user ID.

Try to keep the number of identifiers as small as possible. For example, dividing principals into one group for application users and one for administrators is a good place to start.

Disabling the object authority manager

By default, the OAM is enabled. You can disable it by setting the logical name MQSNOAUT before the queue manager is created, as follows:

```
$ DEFINE/SYSTEM MQSNOAUT TRUE
```

However, if you do this you cannot, in general, restart the OAM later. A much better approach is to have the OAM enabled and ensure that all users and applications have access through an appropriate user ID.

You can also disable the OAM for testing purposes only by removing the authorization service stanza in the queue manager configuration file (qm.ini).

Using the Object Authority Manager commands

The OAM provides a command interface for granting and revoking authority. Before you can use these commands, you must be suitably authorized – your user ID must hold the OpenVMS rights identifier MQM. This identifier should have been set up when you installed the product.

If your user ID holds identifier MQM, you have management authority to the queue manager. This means that you are authorized to issue any MQI request or command from your user ID.

The OAM provides two commands that you can invoke from your OpenVMS DCL to manage the authorizations of users. These are:
- **setmqaut** (Set or reset authority)
- **dspmqaut** (Display authority)

Authority checking occurs in the following calls: MQCONN, MQOPEN, MQPUT1, and MQCLOSE.

Authority checking is only performed at the first instance of any of these calls, and authority is not amended until you reset (that is, close and reopen) the object.

Therefore, any changes made to the authority of an object using **setmqaut** do not take effect until you reset the object.

What you specify when you use the OAM commands

The authority commands apply to the specified queue manager; if you do not specify a queue manager, the default queue manager is used. On these commands, you must specify the object uniquely, that is, you must specify the object name and its type. You also have to specify the principal or identifier name to which the authority applies.

Authorization lists

On the **setmqaut** command you specify a list of authorizations. This is simply a shorthand way of specifying whether authorization is to be granted or revoked, and which resources the authorization applies to. Each authorization in the list is specified as a lowercase keyword, prefixed with a **+** or **–** sign. Use a **+** sign to add the specified authorization or a **–** sign to remove the authorization. You can specify any number of authorizations in a single command. For example:

```
+browse -get +put
```

Using the setmqaut command

Provided you have the required authorization, you can use the **setmqaut** command to grant or revoke authorization of a principal or rights identifier to access a particular object. The following example shows how the **setmqaut** command is used:

```
setmqaut -m "saturn.queue.manager" -t queue -n RED.LOCAL.QUEUE -g GROUPA +browse -get +put
```

In this example:

This term...	Specifies the...
`saturn.queue.manager`	Queue manager name.
`queue`	Object type.
`RED.LOCAL.QUEUE`	Object name.
`GROUPA`	ID of the group to be given the authorizations.
`+browse -get +put`	Authorization list for the specified queue. There must be no spaces between the '+' or '–' signs and the keyword.

The authorization list specifies the authorizations to be given, where:

This term...	Does this...
`+browse`	Adds authorization to browse (MQGET with browse option) messages on the queue.
`-get`	Removes authorization to get (MQGET) messages from the queue.
`+put`	Adds authorization to put (MQPUT) messages on the queue.

This means that applications started with user IDs that hold OpenVMS identifier `GROUPA` have these authorizations.

You can specify one or more principals and, at the same time, one or more identifiers. For example, the following command revokes put authority on the queue `MyQueue` to the principal `FVUSER` and to identifiers `GROUPA` and `GROUPB`.

```
setmqaut -m "saturn.queue.manager" -t queue -n "MyQueue" -p FVUSER -g GROUPA -g GROUPB -put
```

Note: This command also revokes put authority for all rights identifiers held by `FVUSER`, that is, all groups to which `FVUSER` belongs.

For a formal definition of the command and its syntax, see "setmqaut (Set/reset authority)" on page 281.

Authority commands and installable services

The **setmqaut** command takes an additional parameter that specifies the name of the installable service component to which the update applies. You must specify this parameter if you have multiple installable components running at the same time. By default, this is not the case. If the parameter is omitted, the update is made to the first installable service of that type, if one exists. By default, this is the supplied OAM.

Access authorizations

Authorizations defined by the authorization list associated with the **setmqaut** command can be categorized as follows:

- Authorizations related to MQI calls
- Authorization related administration commands
- Context authorizations
- General authorizations, that is, for MQI calls, for commands, or both

Each authorization is specified by a keyword used with the **setmqaut** and **dspmqaut** commands. These are described in "setmqaut (Set/reset authority)" on page 281.

Display authority command

You can use the command **dspmqaut** to view the authorizations that a specific principal or identifier has for a particular object. The flags have the same meaning as those in the **setmqaut** command. Authorization can only be displayed for one identifier or principal at a time. See "dspmqaut (Display authority)" on page 239 for a formal specification of this command.

For example, the following command displays the authorizations that the group `GpAdmin` has to a process definition named `Annuities` on queue manager `QueueMan1`.

```
dspmqaut -m "QueueMan1" -t process -n "Annuities" -g "GpAdmin"
```

The keywords displayed as a result of this command identify the authorizations that are active.

Object Authority Manager guidelines

Some operations are particularly sensitive and should be limited to privileged users. For example,

- Starting and stopping queue managers.
- Accessing certain special queues, such as transmission queues or the command queue SYSTEM.ADMIN.COMMAND.QUEUE.
- Programs that use full MQI context options.
- In general, creating and copying application queues.

User IDs

The special user ID MQM that you created during product installation is intended for use by the product only. It should never be available to non-privileged users.

The user ID used for authorization checks, associated with an MQ process, is the OpenVMS user ID.

Queue manager directories

The directory containing queues and other queue manager data is private to the product. Objects in this directory have OpenVMS user authorizations that relate to their OAM authorizations. However, do not use standard OpenVMS commands to grant or revoke authorizations to MQI resources because:

- MQSeries objects are not necessarily the same as the corresponding system object name. See "Understanding MQSeries file names" on page 19 for more information about this.
- All objects are owned by resource ID MQM.

Queues

The authority to a dynamic queue is based on—but not necessarily the same as—that of the model queue from which it is derived. See note 1 on page 86 for more information.

For alias queues and remote queues, the authorization is that of the object itself, not the queue to which the alias or remote queue resolves. It is, therefore, possible to authorize a user ID to access an alias queue that resolves to a local queue to which the user ID has no access permissions.

You should limit the authority to create queues to privileged users. If you do not, some users may bypass the normal access control simply by creating an alias.

Alternate user authority

Alternate user authority controls whether one user ID can use the authority of another user ID when accessing an MQSeries object. This is essential where a server receives requests from a program and the server wishes to ensure that the program has the required authority for the request. The server may have the required authority, but it needs to know whether the program has the authority for the actions it has requested.

For example:

- A server program running under user ID PAYSERV retrieves a request message from a queue that was put on the queue by user ID USER1.
- When the server program gets the request message, it processes the request and puts the reply back into the reply-to queue specified with the request message.
- Instead of using its own user ID (PAYSERV) to authorize opening the reply-to queue, the server can specify some other user ID, in this case, USER1. In this example, you can use alternate user authority to control whether PAYSERV is allowed to specify USER1 as an alternate user ID when it opens the reply-to queue.

The alternate user ID is specified on the *AlternateUserId* field of the object descriptor.

Note: You can use alternate user IDs on any MQSeries object. Use of an alternate user ID does not affect the user ID used by any other resource managers.

Context authority

Context is information that applies to a particular message and is contained in the message descriptor, MQMD, which is part of the message. The context information comes in two sections:

Identity section
This part specifies who the message came from. It consists of the following fields:
- *UserIdentifier*

- *AccountingToken*
- *ApplIdentityData*

Origin section
: This section specifies where the message came from, and when it was put onto the queue. It consists of the following fields:
- *PutApplType*
- *PutApplName*
- *PutDate*
- *PutTime*
- *ApplOriginData*

Applications can specify the context data when either an MQOPEN or an MQPUT call is made. This data may be generated by the application, it may be passed on from another message, or it may be generated by the queue manager by default. For example, context data can be used by server programs to check the identity of the requester, testing whether the message came from an application, running under an authorized user ID.

A server program can use the *UserIdentifier* to determine the user ID of an alternate user.

You use context authorization to control whether the user can specify any of the context options on any MQOPEN or MQPUT1 call. For information about the context options, see the *MQSeries Application Programming Guide*. For descriptions of the message descriptor fields relating to context, see the *MQSeries Application Programming Reference* book.

Remote security considerations

For remote security, you should consider:

Put authority
: For security across queue managers you can specify the put authority that is used when a channel receives a message sent from another queue manager.

Specify the channel attribute PUTAUT as follows:

DEF Default user ID. This is the user ID that the message channel agent is running under.

CTX The user ID in the message context.

Transmission queues
: Queue managers automatically put remote messages on a transmission queue; no special authority is required for this. However, putting a message directly on a transmission queue requires special authorization; see Table 2 on page 83.

Channel exits
: Channel exits can be used for added security.

For more information, see the *MQSeries Intercommunication* book.

Channel command security

Channel commands can be issued as PCF commands, through the MQAI, MQSC commands, and control commands.

PCF commands

You can issue PCF channel commands by sending a PCF message to the SYSTEM.ADMIN.COMMAND.QUEUE on a remote OpenVMS system. The user ID, as specified in the message descriptor of the PCF message, must hold rights identifier MQM on the target system. These commands are:

- *ChangeChannel*
- *CopyChannel*
- *CreateChannel*
- *DeleteChannel*
- *PingChannel*
- *ResetChannel*
- *StartChannel*
- *StartChannelInitiator*
- *StartChannelListener*
- *StopChannel*
- *ResolveChannel*

See the *MQSeries Programmable System Management* book for the PCF security requirements.

MQSC channel commands

You can issue MQSC channel commands to a remote OpenVMS system either by sending the command directly in a PCF escape message or by issuing the command using **runmqsc** in indirect mode. The user ID as specified in the message descriptor of the associated PCF message must hold rights identifier MQM on the target system. (PCF commands are implicit in MQSC commands issued from **runmqsc** in indirect mode.) These commands are:

- ALTER CHANNEL
- DEFINE CHANNEL
- DELETE CHANNEL
- PING CHANNEL
- RESET CHANNEL
- START CHANNEL
- START CHINIT
- START LISTENER
- STOP CHANNEL
- RESOLVE CHANNEL

For MQSC commands issued from the **runmqsc** command, the user ID in the PCF message is normally that of the current user.

Control commands for channels

For the control commands for channels, the user ID that issues them must hold rights identifier MQM. These commands are:

- **runmqchi** (Run channel initiator)
- **runmqchl** (Run channel)

Understanding the authorization specification tables

The authorization specification tables starting on page 83 define precisely how the authorizations work and the restrictions that apply. The tables apply to these situations:

- Applications that issue MQI calls.
- Administration programs that issue MQSC commands as escape PCFs.
- Administration programs that issue PCF commands.

In this section, the information is presented as a set of tables that specify the following:

Action to be performed
MQI option, MQSC command, or PCF command.

Access control object
Queue, process, or queue manager.

Authorization required
Expressed as an 'MQZAO_' constant.

In the tables, the constants prefixed by MQZAO_ correspond to the keywords in the authorization list for the **setmqaut** command for the particular entity. For example, MQZAO_BROWSE corresponds to the keyword `+browse`; similarly, the keyword MQZAO_SET_ALL_CONTEXT corresponds to the keyword `+setall` and so on. These constants are defined in the header file cmqzc.h, which is supplied with the product. See "What the authorization files contain" on page 89 for more information.

MQI authorizations

An application is only allowed to issue certain MQI calls and options if the user identifier under which it is running (or whose authorizations it is able to assume) has been granted the relevant authorization.

Four MQI calls may require authorization checks: MQCONN, MQOPEN, MQPUT1, and MQCLOSE.

For MQOPEN and MQPUT1, the authority check is made on the name of the object being opened, and not on the name, or names, resulting after a name has been resolved. For example, an application may be granted authority to open an alias queue without having authority to open the base queue to which the alias resolves. The rule is that the check is carried out on the first definition encountered during the process of name resolution that is not a queue-manager alias, unless the queue-manager alias definition is opened directly; that is, its name appears in the *ObjectName* field of the object descriptor. Authority is always needed for the particular object being opened; in some cases additional queue-independent authority—which is obtained through an authorization for the queue-manager object—is required.

Table 2 summarizes the authorizations needed for each call.

Table 2. Security authorization needed for MQI calls

Authorization required for:	Queue object (1)	Process object	Queue manager object	Namelists
MQCONN option	Not applicable	Not applicable	MQZAO_ CONNECT	Not applicable
MQOPEN Option				
MQOO_INQUIRE	MQZAO_INQUIRE (2)	MQZAO_INQUIRE (2)	MQZAO_INQUIRE (2)	MQZAO_INQUIRE (2)
MQOO_BROWSE	MQZAO_BROWSE	Not applicable	No check	Not applicable
MQOO_INPUT_*	MQZAO_INPUT	Not applicable	No check	Not applicable
MQOO_SAVE_ ALL_CONTEXT (3)	MQZAO_INPUT	Not applicable	No check	Not applicable

Authorization specification tables

Table 2. Security authorization needed for MQI calls (continued)

Authorization required for:	Queue object (1)	Process object	Queue manager object	Namelists
MQOO_OUTPUT (Normal queue) (4)	MQZAO_OUTPUT	Not applicable	No check	Not applicable
MQOO_PASS_ IDENTITY_CONTEXT (5)	MQZAO_PASS_ IDENTITY_ CONTEXT	Not applicable	No check	Not applicable
MQOO_PASS_ ALL_CONTEXT (5, 6)	MQZAO_PASS _ALL_CONTEXT	Not applicable	No check	Not applicable
MQOO_SET_ IDENTITY_CONTEXT (5, 6)	MQZAO_SET_ IDENTITY_ CONTEXT	Not applicable	MQZAO_SET_ IDENTITY_ CONTEXT (7)	Not applicable
MQOO_SET_ ALL_CONTEXT (5, 8)	MQZAO_SET_ ALL_CONTEXT	Not applicable	MQZAO_SET_ ALL_CONTEXT (7)	Not applicable
MQOO_OUTPUT (Transmission queue) (9)	MQZAO_SET_ ALL_CONTEXT	Not applicable	MQZAO_SET_ ALL_CONTEXT (7)	Not applicable
MQOO_SET	MQZAO_SET	Not applicable	No check	Not applicable
MQOO_ALTERNATE_ USER_AUTHORITY	(10)	(10)	MQZAO_ ALTERNATE_ USER_ AUTHORITY (10, 11)	(10)
MQPUT1 Option				
MQPMO_PASS_ IDENTITY_CONTEXT	MQZAO_PASS_ IDENTITY_ CONTEXT (12)	Not applicable	No check	Not applicable
MQPMO_PASS_ ALL_CONTEXT	MQZAO_PASS_ ALL_CONTEXT (12)	Not applicable	No check	Not applicable
MQPMO_SET_ IDENTITY_CONTEXT	MQZAO_SET_ IDENTITY_ CONTEXT (12)	Not applicable	MQZAO_SET_ IDENTITY_ CONTEXT (7)	Not applicable
MQPMO_SET_ ALL_CONTEXT	MQZAO_SET_ ALL_CONTEXT (12)	Not applicable	MQZAO_SET_ ALL_CONTEXT (7)	Not applicable
(Transmission queue) (9)	MQZAO_SET_ ALL_CONTEXT	Not applicable	MQZAO_SET_ ALL_CONTEXT (7)	Not applicable
MQPMO_ALTERNATE_ USER_AUTHORITY	(13)	Not applicable	MQZAO _ALTERNATE_ USER_ AUTHORITY (11)	Not applicable
MQCLOSE Option				
MQCO_DELETE	MQZAO_DELETE (14)	Not applicable	Not applicable	Not applicable
MQCO_DELETE_PURGE	MQZAO_DELETE (14)	Not applicable	Not applicable	Not applicable

Specific notes:

1. If a model queue is being opened:

- MQZAO_DISPLAY authority is needed for the model queue, in addition to whatever other authorities (also for the model queue) are required for the open options specified.
- MQZAO_CREATE authority is not needed to create the dynamic queue.
- The user identifier used to open the model queue is automatically granted all of the queue-specific authorities (equivalent to MQZAO_ALL) for the dynamic queue created.

2. Either the queue, process, namelist or queue manager object is checked, depending on the type of object being opened.
3. MQOO_INPUT_* must also be specified. This is valid for a local, model, or alias queue.
4. This check is performed for all output cases, except the case specified in note 9.
5. MQOO_OUTPUT must also be specified.
6. MQOO_PASS_IDENTITY_CONTEXT is also implied by this option.
7. This authority is required for both the queue manager object and the particular queue.
8. MQOO_PASS_IDENTITY_CONTEXT, MQOO_PASS_ALL_CONTEXT, and MQOO_SET_IDENTITY_CONTEXT are also implied by this option.
9. This check is performed for a local or model queue that has a *Usage* queue attribute of MQUS_TRANSMISSION, and is being opened directly for output. It does not apply if a remote queue is being opened (either by specifying the names of the remote queue manager and remote queue, or by specifying the name of a local definition of the remote queue).
10. At least one of MQOO_INQUIRE (for any object type), or (for queues) MQOO_BROWSE, MQOO_INPUT_*, MQOO_OUTPUT, or MQOO_SET must also be specified. The check carried out is as for the other options specified, using the supplied alternate user identifier for the specific-named object authority, and the current application authority for the MQZAO_ALTERNATE_USER_IDENTIFIER check.
11. This authorization allows any *AlternateUserId* to be specified.
12. An MQZAO_OUTPUT check is also carried out, if the queue does not have a *Usage* queue attribute of MQUS_TRANSMISSION.
13. The check carried out is as for the other options specified, using the supplied alternate user identifier for the specific-named queue authority, and the current application authority for the MQZAO_ALTERNATE_USER_IDENTIFIER check.
14. The check is carried out only if both of the following are true:
    - A permanent dynamic queue is being closed and deleted.
    - The queue was not created by the MQOPEN which returned the object handle being used.

Otherwise, there is no check.

General notes:

1. The special authorization MQZAO_ALL_MQI includes all of the following that are relevant to the object type:
   - MQZAO_CONNECT
   - MQZAO_INQUIRE
   - MQZAO_SET
   - MQZAO_BROWSE
   - MQZAO_INPUT
   - MQZAO_OUTPUT
   - MQZAO_PASS_IDENTITY_CONTEXT
   - MQZAO_PASS_ALL_CONTEXT
   - MQZAO_SET_IDENTITY_CONTEXT
   - MQZAO_SET_ALL_CONTEXT
   - MQZAO_ALTERNATE_USER_AUTHORITY
2. MQZAO_DELETE (see note 14 on page 85) and MQZAO_DISPLAY are classed as administration authorizations. They are not therefore included in MQZAO_ALL_MQI.
3. 'No check' means that no authorization checking is carried out.
4. 'Not applicable' means that authorization checking is not relevant to this operation. For example, you cannot issue an MQPUT call to a process object.

Administration authorizations

These authorizations allow a user to issue administration commands. This can be an MQSC command as an escape PCF message or as a PCF command itself. These methods allow a program to send an administration command as a message to a queue manager, for execution on behalf of that user.

Authorizations for MQSC commands in escape PCFs

Table 3 summarizes the authorizations needed for each MQSC command that is contained in Escape PCF.

Table 3. MQSC commands and security authorization needed

(2)Authorization required for:	Queue object	Process object	Queue manager object	Namelists
MQSC command				
ALTER object	MQZAO_CHANGE	MQZAO_CHANGE	MQZAO_CHANGE	MQZAO_CHANGE
CLEAR QLOCAL	MQZAO_CLEAR	Not applicable	Not applicable	Not applicable
DEFINE object NOREPLACE (3)	MQZAO_CREATE (4)	MQZAO_CREATE (4)	Not applicable	MQZAO_CREATE (4)
DEFINE object REPLACE (3, 5)	MQZAO_CHANGE	MQZAO_CHANGE	Not applicable	MQZAO_CHANGE
DELETE object	MQZAO_DELETE	MQZAO_DELETE	Not applicable	MQZAO_DELETE
DISPLAY object	MQZAO_DISPLAY	MQZAO_DISPLAY	MQZAO_DISPLAY	MQZAO_DISPLAY

Specific notes:

1. The user identifier, under which the program (for example, **runmqsc**) which submits the command is running, must also have MQZAO_CONNECT authority to the queue manager.

2. Either the queue, process, namelist or queue manager object is checked, depending on the type of object.
3. For DEFINE commands, MQZAO_DISPLAY authority is also needed for the LIKE object if one is specified, or on the appropriate SYSTEM.DEFAULT.xxx object if LIKE is omitted.
4. The MQZAO_CREATE authority is not specific to a particular object or object type. Create authority is granted for all objects, for a specified queue manager, by specifying an object type of QMGR on the SETMQAUT command.
5. This applies if the object to be replaced does in fact already exist. If it does not, the check is as for DEFINE object NOREPLACE.

General notes:

1. To perform any PCF command, you must have DISPLAY authority on the queue manager.
2. The authority to execute an escape PCF depends on the MQSC command within the text of the escape PCF message.
3. 'Not applicable' means that authorization checking is not relevant to this operation. For example, you cannot issue a CLEAR QLOCAL on a queue manager object.

Authorizations for PCF commands

Table 4 summarizes the authorizations needed for each PCF command.

Table 4. PCF commands and security authorization needed

(2) Authorization required for:	**Queue object**	**Process object**	**Queue manager object**	**Namelists**
PCF command				
Change object	MQZAO_CHANGE	MQZAO_CHANGE	MQZAO_CHANGE	MQZAO_CHANGE
Clear Queue	MQZAO_CLEAR	Not applicable	Not applicable	Not applicable
Copy object (without replace) (3)	MQZAO_CREATE (4)	MQZAO_CREATE (4)	Not applicable	MQZAO_CREATE (4)
Copy object (with replace) (3, 6)	MQZAO_CHANGE	MQZAO_CHANGE	Not applicable	MQZAO_CHANGE
Create object (without replace) (5)	MQZAO_CREATE (4)	MQZAO_CREATE (4)	Not applicable	MQZAO_CREATE (4)
Create object (with replace) (5, 6)	MQZAO_CHANGE	MQZAO_CHANGE	Not applicable	MQZAO_CHANGE
Delete object	MQZAO_DELETE	MQZAO_DELETE	Not applicable	MQZAO_DELETE
Inquire object	MQZAO_DISPLAY	MQZAO_DISPLAY	MQZAO_DISPLAY	MQZAO_DISPLAY
Inquire object names	No check	No check	No check	No check
Reset queue statistics	MQZAO_DISPLAY and MQZAO_CHANGE	Not applicable	Not applicable	Not applicable

Specific notes:

1. The user identifier under which the program submitting the command is running must also have authority to connect to its local queue manager, and to open the command admin queue for output.
2. Either the queue, process, namelist or queue-manager object is checked, depending on the type of object.
3. For Copy commands, MQZAO_DISPLAY authority is also needed for the From object.
4. The MQZAO_CREATE authority is not specific to a particular object or object type. Create authority is granted for all objects, for a specified queue manager, by specifying an object type of QMGR on the SETMQAUT command.
5. For Create commands, MQZAO_DISPLAY authority is also needed for the appropriate SYSTEM.DEFAULT.* object.
6. This applies if the object to be replaced already exists. If it does not, the check is as for Copy or Create without replace.

General notes:

1. To perform any PCF command, you must have DISPLAY authority on the queue manager.
2. The special authorization MQZAO_ALL_ADMIN includes all of the following that are relevant to the object type:
   - MQZAO_CHANGE
   - MQZAO_CLEAR
   - MQZAO_DELETE
   - MQZAO_DISPLAY

   MQZAO_CREATE is not included, because it is not specific to a particular object or object type.
3. 'No check' means that no authorization checking is carried out.
4. 'Not applicable' means that authorization checking is not relevant to this operation. For example, you cannot use a **Clear Queue** command on a process object.

Understanding authorization files

Note: The information in this section is given for problem determination. Under normal circumstances, use authorization commands to view and change authorization information.

MQSeries for Compaq OpenVMS uses a specific file structure to implement security. You do not have to do anything with these files, except to ensure that all the authorization files are themselves secure.

Security is implemented by authorization files. From this perspective, there are three types of authorization:

- Authorizations applying to single object, for example, the authority to put a message on a queue.
- Authorizations applying to a class of objects, for example, the authority to create a queue.
- Authorizations applying across all classes of objects, for example, the authority to perform operations on behalf of different users.

Authorization file paths

The path to an authorization file depends on its type. When you specify an authorization for an object, for example, the queue manager creates the appropriate authorization files. It puts these files into a sub-directory, the path of which is defined by the queue manager name, the type of authorization, and where appropriate, the object name.

Not all authorizations apply directly to instances of objects. For example, the authorization to create an object applies to the class of objects rather than to an individual instance. Also, some authorizations apply across the entire queue manager, for example, alternate user authority means that a user can assume the authorities associated with another user.

Authorization directories

By default, the authorization directories, for a queue manager called `saturn` are:

`MQS_ROOT:[MQM.QMGRS.SATURN.AUTH.QUEUES]`
Authorization files for queues.

`MQS_ROOT:[MQM.QMGRS.SATURN.AUTH.PROCDEF]`
Authorization files for process definitions.

`MQS_ROOT:[MQM.QMGRS.SATURN.AUTH.QMANAGER]`
Authorization files for the queue manager.

`MQS_ROOT:[MQM.QMGRS.SATURN.AUTH]$ACLASS`
Authorizations applying to all classes.

`MQS_ROOT:[MQM.QMGRS.SATURN.AUTH.NAMELIST]`
Authorizations applying to all namelists.

In the object directories, the $CLASS files hold the authorizations related to the entire class.

Note: There is a difference between $CLASS (the authorization file that specifies authorization for a particular class) and $ACLASS (the authorization file that specifies authorizations to all classes).

The paths of the object authorization files are based on those of the object itself, where `AUTH` is inserted ahead of the object type directory. You can use the **dspmqfls** command to display the path to a specified object.

For example, if the name and path of SYSTEM.DEFAULT.LOCAL.QUEUE is:

```
MQS_ROOT:[MQM.QMGRS.SATURN.QUEUES.SYSTEM$DEFAULT$LOCAL$QUEUE]
```

the name and path of the corresponding authorization file is:

```
MQS_ROOT:[MQM.QMGRS.SATURN.AUTH.QUEUES.SYSTEM$DEFAULT$LOCAL$QUEUE]
```

Note: In this case, the actual names of the files associated with the queue are not the same as the name of the queue itself. See “Understanding MQSeries file names” on page 19 for details.

What the authorization files contain

The authorizations of a particular identifier are defined by a set of stanzas in the authorization file. See “Understanding authorization files” on page 88 for more information. The authorizations apply to the object associated with this file. For example:

```
groupb:
   Authority=0x0040007
```

This stanza defines the authority for the identifier `GROUPB`. The authority specification is the union of the individual bit patterns based on the following assignments:

Authorization keyword	Formal name	Hexadecimal Value
connect	MQZAO_CONNECT	0x00000001
browse	MQZAO_BROWSE	0x00000002
get	MQZAO_INPUT	0x00000004
put	MQZAO_OUTPUT	0x00000008
inq	MQZAO_INQUIRE	0x00000010
set	MQZAO_SET	0x00000020
passid	MQZAO_PASS_IDENTITY_CONTEXT	0x00000040
passall	MQZAO_PASS_ALL_CONTEXT	0x00000080
setid	MQZAO_SET_IDENTITY_CONTEXT	0x00000100
setall	MQZAO_SET_ALL_CONTEXT	0x00000200
altusr	MQZAO_ALTERNATE_USER_AUTHORITY	0x00000400
allmqi	MQZAO_ALL_MQI	0x000007FF
crt	MQZAO_CREATE	0x00010000
dlt	MQZAO_DELETE	0x00020000
dsp	MQZAO_DISPLAY	0x00040000
chg	MQZAO_CHANGE	0x00080000
clr	MQZAO_CLEAR	0x00100000
chgaut	MQZAO_AUTHORIZE	0x00800000
alladm	MQZAO_ALL_ADMIN	0x009E0000
none	MQZAO_NONE	0x00000000
all	MQZAO_ALL	0x009E07FF

These definitions are made in the header file cmqzc.h. In the following example, `GROUPB` has been granted authorizations based on the hexadecimal number `0x40007`. This corresponds to:

```
   MQZAO_CONNECT                  0x00000001
   MQZAO_BROWSE                   0x00000002
   MQZAO_INPUT                    0x00000004
   MQZAO_DISPLAY                  0x00040000
                                  ----------
  Authority is:                   0x00040007
```

These access rights mean that anyone in `GROUPB` can issue the MQI calls:

MQCONN
MQGET (with browse)

They also have DISPLAY authority for the object associated with this authorization file.

Class authorization files

The *class authorization files* hold authorizations that relate to the entire class. These files are called “$CLASS” and exist in the same directory as the files for specific objects. The entry MQZAO_CRT in the $CLASS file gives authorization to create an object in the class. This is the only class authority.

All class authorization files

The *all class authorization file* holds authorizations that apply to an entire queue manager. This file is called $ACLASS and exists in the auth subdirectory of the queue manager.

The following authorizations apply to the entire queue manager and are held in the all class authorization file:

The entry...	**Gives authorization to...**
MQZAO_ALTERNATE_USER_AUTHORITY	Assume the identity of another user when interacting with MQSeries objects.
MQZAO_SET_ALL_CONTEXT	Set the context of a message when issuing MQPUT.
MQZAO_SET_IDENTITY_CONTEXT	Set the identity context of a message when issuing MQPUT.

Managing authorization files

Here are some pointers that you need to take into consideration when managing your authorization files:

1. You must ensure that the authorization files are secure and not write-accessible by non-trusted general users. See “Authorizations to authorization files”.
2. To be able to reproduce your file authorizations, ensure that you do at least one of the following:
   - Back up the AUTH subdirectory after any significant updates
   - Retain DCL command files containing the commands used
3. You can copy and edit authorization files. However, you should not normally have to create or repair them manually. Should an emergency occur, you can use the information given here to recover lost or damaged authorization files.

Authorizations to authorization files

Authorization files must be readable by any principal. However, only the system manager and user with the MQM identifier should be allowed to update these files.

The permissions on authorization files, created by the OAM, are:

```
S:RWD, O:RWD, G:RWD, W:R   (ID=MQM, ACCESS=R+W+E+D+C)
```

Do not alter these permissions without reviewing carefully whether there are any security exposures.

To alter authorizations using the command supplied with MQSeries for Compaq OpenVMS, your process must have the MQM rights identifier.

Chapter 8. The MQSeries dead-letter queue handler

A *dead-letter queue* (DLQ), sometimes referred to as an *undelivered-message queue*, is a holding queue for messages that cannot be delivered to their destination queues. Every queue manager in a network should have an associated DLQ.

Messages can be put on the DLQ by queue managers, by message channel agents (MCAs), and by applications. All messages on the DLQ should be prefixed with a *dead-letter header* structure, MQDLH. Messages put on the DLQ by a queue manager or by a message channel agent always have an MQDLH; applications putting messages on the DLQ are strongly recommended to supply an MQDLH. The *Reason* field of the MQDLH structure contains a reason code that identifies why the message is on the DLQ.

In all MQSeries environments, there should be a routine that runs regularly to process messages on the DLQ. MQSeries supplies a default routine, called the *dead-letter queue handler* (the DLQ handler), which you invoke using the **runmqdlq** command.

Instructions for processing messages on the DLQ are supplied to the DLQ handler by means of a user-written *rules table*. That is, the DLQ handler matches messages on the DLQ against entries in the rules table: when a DLQ message matches an entry in the rules table, the DLQ handler performs the action associated with that entry.

This chapter contains the following sections:
- "Invoking the DLQ handler"
- "The DLQ handler rules table" on page 94
- "How the rules table is processed" on page 100
- "An example DLQ handler rules table" on page 102

Invoking the DLQ handler

You invoke the DLQ handler using the **runmqdlq** command. You can name the DLQ you want to process and the queue manager you want to use in two ways:
- As parameters to **runmqdlq** from the command prompt. For example:

```
runmqdlq ABC1.DEAD.LETTER.QUEUE ABC1.QUEUE.MANAGER < qrule.rul
```

- In the rules table. For example:

```
INPUTQ(ABC1.DEAD.LETTER.QUEUE) INPUTQM(ABC1.QUEUE.MANAGER)
```

The above examples apply to the DLQ called ABC1.DEAD.LETTER.QUEUE, owned by the queue manager ABC1.QUEUE.MANAGER.

If you do not specify the DLQ or the queue manager as shown above, the default queue manager for the installation is used along with the DLQ belonging to that queue manager.

The **runmqdlq** command takes its input from `SYS$INPUT`: you associate the rules table with **runmqdlq** by redirecting `SYS$INPUT` from the rules table.

Attention: Running the DLQ handler without redirecting SYS$INPUT to a rule file causes the DLQ handler to loop.

In order to run the DLQ handler, you must be authorized to access both the DLQ itself and any message queues to which messages on the DLQ are forwarded. Furthermore, if the DLQ handler is to be able to put messages on queues with the authority of the user ID in the message context, you must be authorized to assume the identity of other users.

For more information about the **runmqdlq** command, see "runmqdlq (Run dead-letter queue handler)" on page 271.

The sample DLQ handler, amqsdlq

In addition to the DLQ handler invoked using the **runmqdlq** command, MQSeries provides the source of a sample DLQ handler, amqsdlq, whose function is similar to that provided via **runmqdlq**. The sources are provided as templates only and must be customized to provide a DLQ handler that meets specific, local requirements. For example, you might decide that you want a DLQ handler that can process messages without dead-letter headers. (Both the default DLQ handler and the sample, amqsdlq, process only those messages on the DLQ that begin with a dead-letter header, MQDLH. Messages that do not begin with an MQDLH are identified as being in error, and remain on the DLQ indefinitely.)

The source of amqsdlq is supplied in the directory:

```
[.DLQ], under MQS_EXAMPLES
```

and the compiled version is supplied in the directory:

```
[.BIN], under MQS_EXAMPLES
```

The DLQ handler rules table

The DLQ handler rules table defines how the DLQ handler is to process messages that arrive on the DLQ. There are two types of entry in a rules table:
- The first entry in the table, which is optional, contains *control data*.
- All other entries in the table are *rules* for the DLQ handler to follow. Each rule consists of a *pattern* (a set of message characteristics) that a message is matched against, and an *action* to be taken when a message on the DLQ matches the specified pattern. There must be at least one rule in a rules table.

Each entry in the rules table comprises one or more keywords.

Control data

This section describes the keywords that you can include in a control-data entry in a DLQ handler rules table. Please note the following:
- The default value for a keyword, if any, is underlined.
- The vertical line (|) separates alternatives, only one of which can be specified.
- All keywords are optional.

INPUTQ (*QueueName***|'_')**
Allows you to name the DLQ you want to process:

1. If you specify an INPUTQ value as a parameter to the **runmqdlq** command, this overrides any INPUTQ value in the rules table.
2. If you do not specify an INPUTQ value as a parameter to the **runmqdlq** command but you *do* specify a value in the rules table, the INPUTQ value in the rules table is used.
3. If no DLQ is specified or you specify INPUTQ(' ') in the rules table, the name of the DLQ belonging to the queue manager whose name is supplied as a parameter to the **runmqdlq** command is used.
4. If you do not specify an INPUTQ value as a parameter to the **runmqdlq** command or as a value in the rules table, the DLQ belonging to the queue manager named on the INPUTQM keyword in the rules table is used.

INPUTQM (*QueueManagerName***|'_')**
Allows you to name the queue manager that owns the DLQ named on the INPUTQ keyword:

1. If you specify an INPUTQM value as a parameter to the **runmqdlq** command, this overrides any INPUTQM value in the rules table.
2. If you do not specify an INPUTQM value as a parameter to the **runmqdlq** command, the INPUTQM value in the rules table is used.
3. If no queue manager is specified or you specify INPUTQM(' ') in the rules table, the default queue manager for the installation is used.

RETRYINT (*Interval***|60)**
Is the interval, in seconds, at which the DLQ handler should attempt to reprocess messages on the DLQ that could not be processed at the first attempt, and for which repeated attempts have been requested. By default, the retry interval is 60 seconds.

WAIT (YES|NO|*nnn***)**
Indicates whether the DLQ handler should wait for further messages to arrive on the DLQ when it detects that there are no further messages that it can process.

YES	Causes the DLQ handler to wait indefinitely.
NO	Causes the DLQ handler to terminate when it detects that the DLQ is either empty or contains no messages that it can process.
nnn	Causes the DLQ handler to wait for *nnn* seconds for new work to arrive before terminating, after it detects that the queue is either empty or contains no messages that it can process.

You are recommended to specify WAIT (YES) for busy DLQs, and WAIT (NO) or WAIT (*nnn*) for DLQs that have a low level of activity. If the DLQ handler is allowed to terminate, you are recommended to reinvoke it by means of triggering.

As an alternative to including control data in the rules table, you can supply the names of the DLQ and its queue manager as input parameters of the **runmqdlq** command. If any value is specified both in the rules table and on input to the **runmqdlq** command, the value specified on the **runmqdlq** command takes precedence.

Note: If a control-data entry is included in the rules table, it ***must*** be the first entry in the table.

Rules (patterns and actions)

Figure 8 shows an example rule from a DLQ handler rules table.

```
PERSIST(MQPER_PERSISTENT) REASON(MQRC_PUT_INHIBITED) +
 ACTION(RETRY) RETRY(3)
```

Figure 8. An example rule from a DLQ handler rules table. This rule instructs the DLQ handler to make 3 attempts to deliver to its destination queue any persistent message that was put on the DLQ because MQPUT and MQPUT1 were inhibited.

All keywords that you can use on a rule are described in the remainder of this section. Please note the following:

- The default value for a keyword, if any, is underlined. For most keywords, the default value is * (asterisk), which matches any value.
- The vertical line (|) separates alternatives, only one of which can be specified.
- All keywords except ACTION are optional.

This section begins with a description of the pattern-matching keywords (those against which messages on the DLQ are matched), and then describes the action keywords (those that determine how the DLQ handler is to process a matching message).

The pattern-matching keywords

The pattern-matching keywords, which you use to specify values against which messages on the DLQ are matched, are described below. All pattern-matching keywords are optional.

APPLIDAT (*ApplIdentityData***|<u>*</u>)**
: Is the *ApplIdentityData* value specified in the message descriptor, MQMD, of the message on the DLQ.

APPLNAME (*PutApplName***|<u>*</u>)**
: Is the name of the application that issued the MQPUT or MQPUT1 call, as specified in the *PutApplName* field of the message descriptor, MQMD, of the message on the DLQ.

APPLTYPE (*PutApplType***|<u>*</u>)**
: Is the *PutApplType* value specified in the message descriptor, MQMD, of the message on the DLQ.

DESTQ (*QueueName***|<u>*</u>)**
: Is the name of the message queue for which the message is destined.

DESTQM (*QueueManagerName***|<u>*</u>)**
: Is the name of the queue manager of the message queue for which the message is destined.

FEEDBACK (*Feedback***|<u>*</u>)**
: When the *MsgType* value is MQFB_REPORT, *Feedback* describes the nature of the report.

  Symbolic names can be used. For example, you can use the symbolic name MQFB_COA to identify those messages on the DLQ that require confirmation of their arrival on their destination queues.

FORMAT (*Format***|<u>*</u>)**
: Is the name that the sender of the message uses to describe the format of the message data.

MSGTYPE (*MsgType***|*)**
Is the message type of the message on the DLQ.

Symbolic names can be used. For example, you can use the symbolic name MQMT_REQUEST to identify those messages on the DLQ that require replies.

PERSIST (*Persistence***|*)**
Is the persistence value of the message. (The persistence of a message determines whether it survives restarts of the queue manager.)

Symbolic names can be used. For example, you can use the symbolic name MQPER_PERSISTENT to identify those messages on the DLQ that are persistent.

REASON (*ReasonCode***|*)**
Is the reason code that describes why the message was put to the DLQ.

Symbolic names can be used. For example, you can use the symbolic name MQRC_Q_FULL to identify those messages placed on the DLQ because their destination queues were full.

REPLYQ (*QueueName***|*)**
Is the name of the reply-to queue specified in the message descriptor, MQMD, of the message on the DLQ.

REPLYQM (*QueueManagerName***|*)**
Is the name of the queue manager of the reply-to queue, as specified in the message descriptor, MQMD, of the message on the DLQ.

USERID (*UserIdentifier***|*)**
Is the user ID of the user who originated the message on the DLQ, as specified in the message descriptor, MQMD.

The action keywords

The action keywords, which you use to describe how a matching message is to be processed, are described below.

ACTION (DISCARD|IGNORE|RETRY|FWD)
Is the action to be taken for any message on the DLQ that matches the pattern defined in this rule.

DISCARD
Causes the message to be deleted from the DLQ.

IGNORE
Causes the message to be left on the DLQ.

RETRY
Causes the DLQ handler to try again to put the message on its destination queue.

FWD Causes the DLQ handler to forward the message to the queue named on the FWDQ keyword.

The ACTION keyword must be specified. The number of attempts made to implement an action is governed by the RETRY keyword. The interval between attempts is controlled by the RETRYINT keyword of the control data.

FWDQ (*QueueName***|&DESTQ|&REPLYQ)**
Is the name of the message queue to which the message should be forwarded when ACTION (FWD) is requested.

QueueName
Is the name of a message queue. FWDQ(' ') is not valid.

&DESTQ
Causes the queue name to be taken from the *DestQName* field in the MQDLH structure.

&REPLYQ
Causes the name to be taken from the *ReplyToQ* field in the message descriptor, MQMD.

To avoid error messages when a rule specifying FWDQ (&REPLYQ) matches a message with a blank *ReplyToQ* field, you can specify REPLYQ (?*) in the message pattern.

FWDQM (*QueueManagerName***|&DESTQM|&REPLYQM|'_')**
Identifies the queue manager of the queue to which a message is to be forwarded.

QueueManagerName
Is the name of the queue manager of the queue to which a message is to be forwarded when ACTION (FWD) is requested.

&DESTQM
Causes the queue manager name to be taken from the *DestQMgrName* field in the MQDLH structure.

&REPLYQM
Causes the name to be taken from the *ReplyToQMgr* field in the message descriptor, MQMD.

' ' FWDQM(' '), which is the default value, identifies the local queue manager.

HEADER (YES|NO)
Specifies whether the MQDLH should remain on a message for which ACTION (FWD) is requested. By default, the MQDLH remains on the message. The HEADER keyword is not valid for actions other than FWD.

PUTAUT (DEF|CTX)
Defines the authority with which messages should be put by the DLQ handler:

DEF Causes messages to be put with the authority of the DLQ handler itself.

CTX Causes the messages to be put with the authority of the user ID in the message context. If you specify PUTAUT (CTX), you must be authorized to assume the identity of other users.

RETRY (*RetryCount***|1)**
Is the number of times, in the range 1–999,999,999, that an action should be attempted (at the interval specified on the RETRYINT keyword of the control data).

Note: The count of attempts made by the DLQ handler to implement any particular rule is specific to the current instance of the DLQ handler; the count does not persist across restarts. If the DLQ handler is restarted, the count of attempts made to apply a rule is reset to zero.

Rules table conventions

The rules table must adhere to the following conventions regarding its syntax, structure, and contents:

- A rules table must contain at least one rule.
- Keywords can occur in any order.

- A keyword can be included once only in any rule.
- Keywords are not case sensitive.
- A keyword and its parameter value must be separated from other keywords by at least one blank or comma.
- Any number of blanks can occur at the beginning or end of a rule, and between keywords, punctuation, and values.
- Each rule must begin on a new line.
- For reasons of portability, the significant length of a line should not be greater than 72 characters.
- Use the plus sign (+) as the last nonblank character on a line to indicate that the rule continues from the first nonblank character in the next line. Use the minus sign (–) as the last nonblank character on a line to indicate that the rule continues from the start of the next line. Continuation characters can occur within keywords and parameters.
- Comment lines, which begin with an asterisk (*), can occur anywhere in the rules table.
- Blank lines are ignored.
- Each entry in the DLQ handler rules table comprises one or more keywords and their associated parameters. The parameters must follow these syntax rules:
  - Each parameter value must include at least one significant character. The delimiting quotation marks in quoted values are not considered significant. For example, these parameters are valid:

FORMAT('ABC')	3 significant characters
FORMAT(ABC)	3 significant characters
FORMAT('A')	1 significant character
FORMAT(A)	1 significant character
FORMAT(' ')	1 significant character

    These parameters are invalid because they contain no significant characters:
    FORMAT('')
    FORMAT( )
    FORMAT()
    FORMAT
  - Wildcard characters are supported: you can use the question mark (?) in place of any single character, except a trailing blank; you can use the asterisk (*) in place of zero or more adjacent characters. The asterisk (*) and the question mark (?) are ***always*** interpreted as wildcard characters in parameter values.
  - Wildcard characters cannot be included in the parameters of these keywords: ACTION, HEADER, RETRY, FWDQ, FWDQM, and PUTAUT.
  - Trailing blanks in parameter values, and in the corresponding fields in the message on the DLQ, are not significant when performing wildcard matches. However, leading and embedded blanks within strings in quotation marks are significant to wildcard matches.
  - Numeric parameters cannot include the question mark (?) wildcard character. The asterisk (*) can be used in place of an entire numeric parameter, but cannot be included as part of a numeric parameter. For example, these are valid numeric parameters:

MSGTYPE(2)	Only reply messages are eligible
MSGTYPE(*)	Any message type is eligible
MSGTYPE('*')	Any message type is eligible

However, `MSGTYPE('2*')` is not valid, because it includes an asterisk (*) as part of a numeric parameter.

– Numeric parameters must be in the range 0–999,999,999. If the parameter value is in this range, it is accepted, even if it is not currently valid in the field to which the keyword relates. Symbolic names can be used for numeric parameters.

– If a string value is shorter than the field in the MQDLH or MQMD to which the keyword relates, the value is padded with blanks to the length of the field. If the value, excluding asterisks, is longer than the field, an error is diagnosed. For example, these are all valid string values for an 8-character field:

'ABCDEFGH'	8 characters
'A*C*E*G*I'	5 characters excluding asterisks
'*A*C*E*G*I*K*M*O*'	8 characters excluding asterisks

– Strings that contain blanks, lowercase characters, or special characters other than period (.), forward slash (/), underscore (_), and percent sign (%) must be enclosed in single quotation marks. Lowercase characters not enclosed in quotation marks are folded to uppercase. If the string includes a quotation, two single quotation marks must be used to denote both the beginning and the end of the quotation. When the length of the string is calculated, each occurrence of double quotation marks is counted as a single character.

How the rules table is processed

The DLQ handler searches the rules table for a rule whose pattern matches a message on the DLQ. The search begins with the first rule in the table, and continues sequentially through the table. When a rule with a matching pattern is found, the action from that rule is attempted. The DLQ handler increments the retry count for a rule by 1 whenever it attempts to apply that rule. If the first attempt fails, the attempt is repeated until the count of attempts made matches the number specified on the RETRY keyword. If all attempts fail, the DLQ handler searches for the next matching rule in the table.

This process is repeated for subsequent matching rules until an action is successful. When each matching rule has been attempted the number of times specified on its RETRY keyword, and all attempts have failed, ACTION (IGNORE) is assumed. ACTION (IGNORE) is also assumed if no matching rule is found.

Notes:

1. Matching rule patterns are sought only for messages on the DLQ that begin with an MQDLH. Messages that do not begin with an MQDLH are reported periodically as being in error, and remain on the DLQ indefinitely.
2. All pattern keywords can be allowed to default, such that a rule may consist of an action only. Note, however, that action-only rules are applied to all messages on the queue that have MQDLHs and that have not already been processed in accordance with other rules in the table.
3. The rules table is validated when the DLQ handler is started, and errors are flagged at that time. You can make changes to the rules table at any time, but those changes do not come into effect until the DLQ handler is restarted.
4. The DLQ handler does not alter the content of messages, of the MQDLH, or of the message descriptor. The DLQ handler always puts messages to other queues with the message option MQPMO_PASS_ALL_CONTEXT.
5. The DLQ handler opens the DLQ with the MQOO_INPUT_AS_Q_DEF option.

6. Multiple instances of the DLQ handler could run concurrently against the same queue, using the same rules table. However, it is more usual for there to be a one-to-one relationship between a DLQ and a DLQ handler.

Ensuring that all DLQ messages are processed

The DLQ handler keeps a record of all messages on the DLQ that have been seen but not removed. If you use the DLQ handler as a filter to extract a small subset of the messages from the DLQ, the DLQ handler still has to keep a record of those messages on the DLQ that it did not process. Also, the DLQ handler cannot guarantee that new messages arriving on the DLQ will be seen, even if the DLQ is defined as first-in-first-out (FIFO). Therefore, if the queue is not empty, a periodic rescan of the DLQ is performed to check all messages. For these reasons, you should try to ensure that the DLQ contains as few messages as possible; if messages that cannot be discarded or forwarded to other queues (for whatever reason) are allowed to accumulate on the queue, the workload of the DLQ handler increases and the DLQ itself is in danger of filling up.

You can take specific measures to enable the DLQ handler to empty the DLQ. For example, try not to use ACTION (IGNORE), which simply leaves messages on the DLQ. (Remember that ACTION (IGNORE) is assumed for messages that are not explicitly addressed by other rules in the table.) Instead, for those messages that you would otherwise ignore, use an action that moves the messages to another queue. For example:

```
ACTION (FWD) FWDQ (IGNORED.DEAD.QUEUE) HEADER (YES)
```

Similarly, the final rule in the table should be a catchall to process messages that have not been addressed by earlier rules in the table. For example, the final rule in the table could be something like this:

```
ACTION (FWD) FWDQ (REALLY.DEAD.QUEUE) HEADER (YES)
```

This action causes messages that fall through to the final rule in the table to be forwarded to the queue `REALLY.DEAD.QUEUE`, where they can be processed manually. If you do not have such a rule, messages are likely to remain on the DLQ indefinitely.

An example DLQ handler rules table

Here is an example rules table that contains a single control-data entry and several rules:

```
*************************************************************************
*       An example rules table for the runmqdlq command                 *
*************************************************************************
* Control data entry
* ------------------
* If no queue manager name is supplied as an explicit parameter to
* runmqdlq, use the default queue manager for the machine.
* If no queue name is supplied as an explicit parameter to runmqdlq,
* use the DLQ defined for the local queue manager.
*
inputqm(' ')  inputq(' ')

* Rules
* -----
* We include rules with ACTION (RETRY) first to try to
* deliver the message to the intended destination.

* If a message is placed on the DLQ because its destination
* queue is full, attempt to forward the message to its
* destination queue. Make 5 attempts at approximately
* 60-second intervals (the default value for RETRYINT).

REASON(MQRC_Q_FULL) ACTION(RETRY) RETRY(5)

* If a message is placed on the DLQ because of a put inhibited
* condition, attempt to forward the message to its
* destination queue. Make 5 attempts at approximately
* 60-second intervals (the default value for RETRYINT).

REASON(MQRC_PUT_INHIBITED) ACTION(RETRY) RETRY(5)

* The AAAA corporation are always sending messages with incorrect
* addresses. When we find a request from the AAAA corporation,
* we return it to the DLQ (DEADQ) of the reply-to queue manager
* (&REPLYQM).
* The AAAA DLQ handler attempts to redirect the message.

MSGTYPE(MQMT_REQUEST) REPLYQM(AAAA.*) +
  ACTION(FWD) FWDQ(DEADQ) FWDQM(&REPLYQM)

* The BBBB corporation never do things by half measures. If
* the queue manager BBBB.1 is unavailable, try to
* send the message to BBBB.2

DESTQM(bbbb.1) +
  action(fwd) fwdq(&DESTQ) fwdqm(bbbb.2) header(no)

* The CCCC corporation considers itself very security
* conscious, and believes that none of its messages
* will ever end up on one of our DLQs.
* Whenever we see a message from a CCCC queue manager on our
* DLQ, we send it to a special destination in the CCCC organization
* where the problem is investigated.

REPLYQM(CCCC.*) +
  ACTION(FWD) FWDQ(ALARM) FWDQM(CCCC.SYSTEM)

* Messages that are not persistent run the risk of being
* lost when a queue manager terminates. If an application
* is sending nonpersistent messages, it should be able
* to cope with the message being lost, so we can afford to
* discard the message.
```

```
PERSIST(MQPER_NOT_PERSISTENT) ACTION(DISCARD)

* For performance and efficiency reasons, we like to keep
* the number of messages on the DLQ small.
* If we receive a message that has not been processed by
* an earlier rule in the table, we assume that it
* requires manual intervention to resolve the problem.
* Some problems are best solved at the node where the
* problem was detected, and others are best solved where
* the message originated. We don't have the message origin,
* but we can use the REPLYQM to identify a node that has
* some interest in this message.
* Attempt to put the message onto a manual intervention
* queue at the appropriate node. If this fails,
* put the message on the manual intervention queue at
* this node.

REPLYQM('?*') +
  ACTION(FWD) FWDQ(DEADQ.MANUAL.INTERVENTION) FWDQM(&REPLYQM)

ACTION(FWD) FWDQ(DEADQ.MANUAL.INTERVENTION)
```

Chapter 9. Instrumentation events

You can use the MQSeries instrumentation events to monitor the operation of queue managers. This chapter provides a short introduction to instrumentation events. For a more complete description, see the section on instrumentation events in the *MQSeries Programmable System Management* book.

What are instrumentation events?

Instrumentation events cause special messages, called *event messages*, to be generated whenever the queue manager detects a predefined set of conditions. For example, the following conditions give rise to a *Queue Full* event:

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

Other conditions that can give rise to instrumentation events include:

- A threshold limit for the number of messages on a queue being reached.
- A queue not being serviced within a specified time period.
- A channel instance being started or stopped.
- In an MQSeries for Compaq OpenVMS system, an application attempting to open a queue specifying a user ID that is not authorized.

With the exception of channel events, all instrumentation events must be enabled before they can be generated.

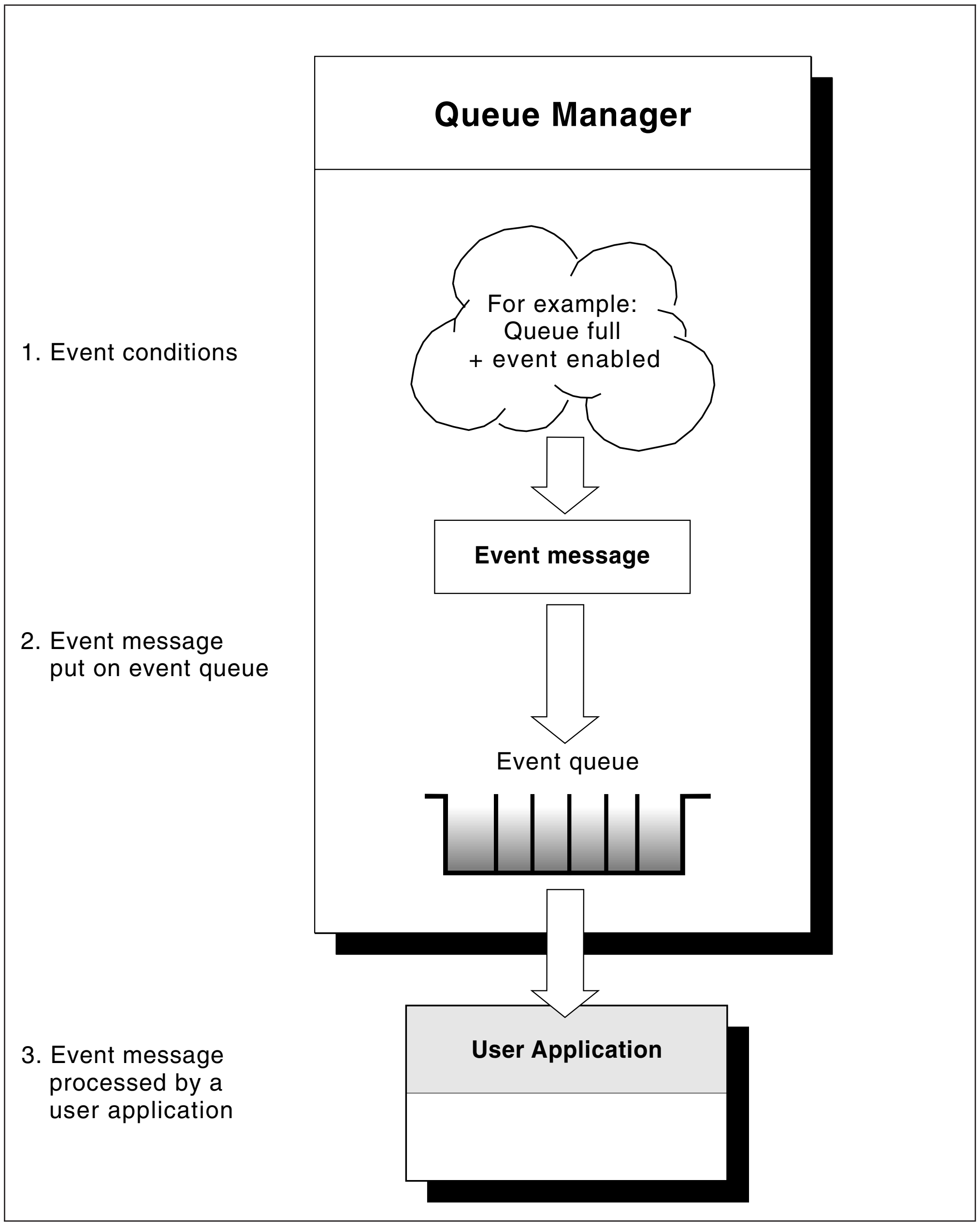


Figure 9. Understanding instrumentation events. When a queue manager detects that the conditions for an event have been met, it puts an event message on the appropriate event queue.

The event message, which contains information about the conditions giving rise to the event, is put onto an *event queue*. An application can retrieve the event message from this queue for analysis.

Why use events?

If you specify your event queues as remote queues, you can put all the event queues on a single queue manager (for those nodes that support instrumentation events). You can then use the events generated to monitor a network of queue managers from a single node. Figure 10 on page 107 illustrates this.

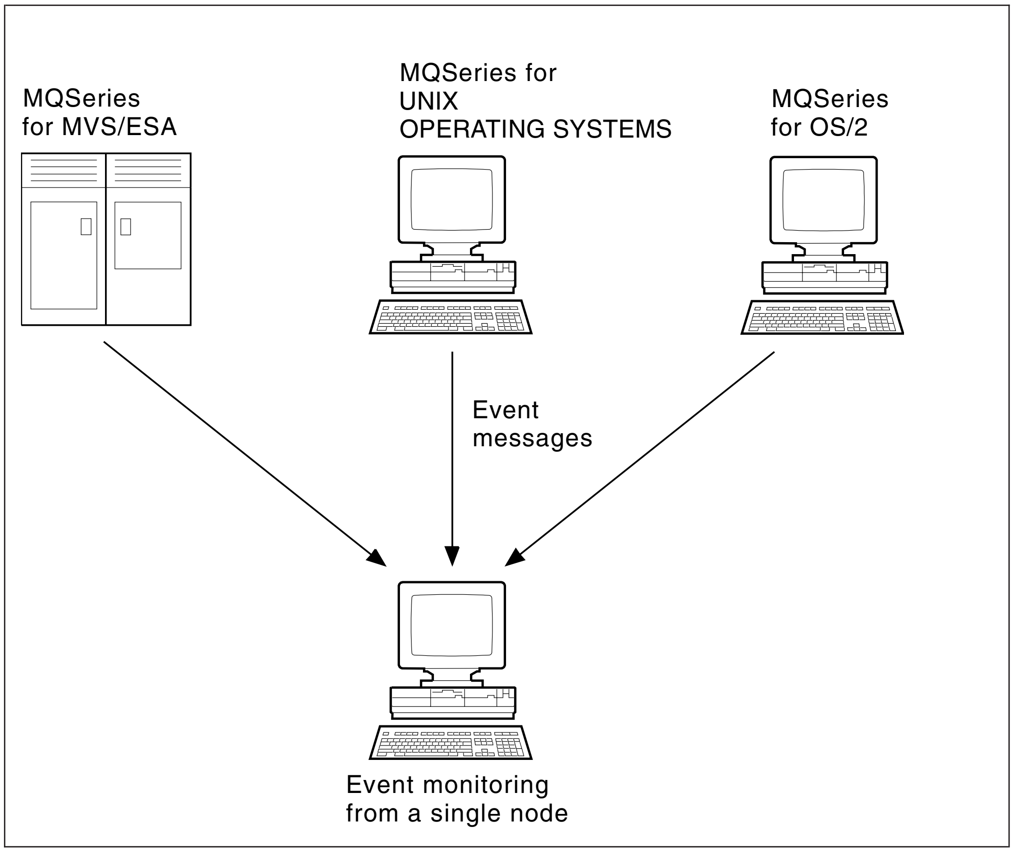

MQSeries for MVS/ESA
MQSeries for UNIX OPERATING SYSTEMS
MQSeries for OS/2
Event messages
Event monitoring from a single node

Figure 10. Monitoring queue managers across different platforms, on a single node

Types of events

MQSeries 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 or, following a get, the queue was not serviced within a predefined time limit.

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.

Trigger events

When we discuss triggering in this and other MQSeries books, we 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, a queue can be configured to generate a trigger event each time a message arrives. (The conditions for trigger events and instrumentation events are quite different.)

A trigger event causes a trigger message to be put on an initiation queue and, optionally, an application program is started.

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 a description of event message formats, see the *MQSeries Programmable System Management* book.

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.PERFM.EVENT	Performance events
SYSTEM.ADMIN.CHANNEL.EVENT	Channel events

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

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 an application that performs some administration tasks automatically.

Enabling and disabling events

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

- MQSC commands. For more information, see the *MQSeries Command Reference* book.

- PCF commands for queue managers on UNIX systems, OpenVMS systems, and OS/2. For more information, see the *MQSeries Programmable System Management* book.
- MQAI commands. For more information, see the *MQSeries Administration Interface Programming Guide and Reference* book.

Enabling an event depends on the category of the event:
- Queue manager events are enabled by setting attributes on the queue manager.
- Performance events as a whole must be enabled on the queue manager, or no performance events can occur. You 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 example, a queue depth high limit.
- Channel events occur automatically; they do not need to be enabled. If you do not want to monitor channel events, you can put-inhibit the channel event queue.

Event messages

Event messages contain information relating to the origin of an event, including the type of event, the name of the application that caused the event, and for performance events a short statistics summary for the queue.

The format of event messages is similar to that of PCF response messages. The message data can be retrieved from them by user-written administration programs using the data structures described in the *MQSeries Programmable System Management* book.

Chapter 10. Transactional support

The *MQSeries Application Programming Guide* contains a complete introduction to the subject of this chapter. A brief introduction only is provided here.

An application program can group a set of updates into a *unit of work*. These updates are usually logically related and must all be successful for data integrity to be preserved. If one update succeeded while another failed then data integrity would be lost.

A unit of work **commits** when it completes successfully. At this point all updates made within that unit of work are made permanent or irreversible. If the unit of work fails then all updates are instead *backed out*. *Syncpoint coordination* is the process by which units of work are either committed or backed out with integrity.

A *local* unit of work is one in which the only resources updated are those of the MQSeries queue manager. Here syncpoint coordination is provided by the queue manager itself using a single-phase commit process.

A *global* unit of work is one in which resources belonging to other resource managers, such as XA-compliant databases, are also updated. Here, a two-phase commit procedure must be used and the unit of work may be coordinated by the queue manager itself.

In summary, queue manager resources can be updated as part of local or global units of work:

Local unit of work
Use local units of work when the only resources to be updated are those of the MQSeries queue manager. Updates are committed using the MQCMIT verb or backed out using MQBACK.

Global unit of work
Use global units of work when you also need to include updates to XA-compliant database managers. Here the coordination may be internal or external to the queue manager.

Queue manager coordination
Global units of work are started using the MQBEGIN verb and then committed using MQCMIT or backed out using MQBACK. A two-phase commit process is used whereby XA-compliant resource managers such as Oracle® are firstly all asked to prepare to commit. Only if all are prepared successfully will they then be asked to commit. If any resource manager signals that it cannot prepare to commit, each will be asked to back out instead.

External coordination
Here the coordination is performed by an XA-compliant transaction manager such as IBM CICS®, Transarc Encina, or BEA Tuxedo. Units of work are started and committed under control of the transaction manager. The MQBEGIN, MQCMIT and MQBACK verbs are unavailable.

This chapter describes how to enable support for global units of work (support for local units of work does not need to be specifically enabled).

Transactional support

It contains these sections:
- "Database coordination"
- "Oracle configuration" on page 115
- "Administration tasks" on page 119

Database coordination

When the queue manager coordinates global units of work itself it becomes possible to integrate database updates within MQ units of work. That is, a mixed MQI and SQL application can be written, and the MQCMIT and MQBACK verbs can be used to commit or roll back the changes to the queues and databases together.

The queue manager achieves this using a two-phase commit protocol. When a unit of work is to be committed, the queue manager first asks each participating database manager whether it is prepared to commit its updates. Only if all of the participants, including the queue manager itself, are prepared to commit, are all of the queue and database updates committed. If any participant cannot prepare its updates, the unit of work is backed out instead.

Full recovery support is provided if the queue manager loses contact with any of the database managers during the commit protocol. If a database manager becomes unavailable while it is in doubt, that is, it has been called to prepare but has yet to receive a commit or back out decision, the queue manager remembers the outcome of the unit of work until it has been successfully delivered. Similarly, if the queue manager terminates with incomplete commit operations outstanding, these are remembered over queue manager restart.

The MQI verb, MQBEGIN, must be used to denote units of work that are also to involve database updates. The *MQSeries Application Programming Guide* identifies sample programs that make MQSeries and database updates within the same unit of work.

The queue manager communicates with the database managers using the XA interface as described in *X/Open Distributed Transaction Processing: The XA Specification* (ISBN 1 872630 24 3). This means that the queue manager can communicate to database managers that also adhere to this standard. Such database managers are known as *XA-compliant* database managers.

Table 5 identifies XA-compliant database managers that are supported by the MQSeries Version 5.1 products.

Table 5. XA-compliant relational databases

MQSeries product	DB2®	Oracle	Sybase®
MQSeries for AIX	Yes	Yes	Yes
MQSeries for HP-UX	Yes	Yes	No
MQSeries for Sun Solaris	Yes	Yes	Yes
MQSeries for Compaq OpenVMS Alpha	No	Yes	No
MQSeries for Windows NT and Windows 2000	Yes	No	Yes

Restrictions

The following restrictions apply to the database coordination support:

- The ability to coordinate database updates within MQSeries units of work is **not** supported in an MQI client application.
- The MQI updates and database updates must be made on the same queue manager server machine.
- The database server may reside on a different machine from the queue manager server. In this case, the database needs to be accessed via an XA-compliant client feature provided by the database manager itself.
- Although the queue manager itself is XA-compliant, it is not possible to configure another queue manager as a participant in global units of work. This is because only one connection at a time can be supported.

Database connections

An application that establishes a standard connection to the queue manager will be associated with a thread in a separate local queue manager agent process. When the application issues MQBEGIN then both it and the agent process will need to connect to the databases that are to be involved in the unit of work. The database connections are maintained while the application remains connected to the queue manager. This is an important consideration if the database only supports a limited number of users or connections.

One method of reducing the number of connections is for the application to use the MQCONNX call to request a fastpath binding. In this case the application and the local queue manager agent become the same process and consequently can share a single database connection. Before you do this, consult the *MQSeries Application Programming Guide* for a list of restrictions that apply to fastpath applications.

Configuring database managers

There are several tasks that you must perform before a database manager can participate in global units of works coordinated by the queue manager:

1. Create an *XA switch load file*[1] for the database manager.
2. Define the database manager in the queue manager's configuration file, `qm.ini`.

   Various items, including the name of the switch load file, must be defined in `qm.ini`.

Creating switch load files

Instructions for creating switch load files for the supported database managers are provided in "Creating the Oracle switch load file" on page 115.

Refer to your MQSeries installation documentation for more information about the installation procedure.

The sample source modules that are used to produce the switch load files all contain a single function called MQStart. When the switch load file is loaded, the queue manager calls this function and it returns the address of a structure called an XA switch. The switch load file is linked to a library provided by the database manager, which enables MQSeries to call that database manager.

1. An XA switch load file is a dynamically loaded object that enables the queue manager and the database manager to communicate with each other.

Configuring database managers

The sample source modules used to build the switch load files for Oracle are oraswit.c.

Defining database managers

When you have created a switch load file for your database manager, you must specify its location to your queue manager. This is done in the queue manager's `qm.ini` file in the `XAResourceManager` stanza.

You need to add an `XAResourceManager` stanza for each database manager that your queue manager is going to coordinate.

The attributes of the `XAResourceManager` stanza are as follows.

Name=name
User-chosen string that identifies the database manager instance.

The name is mandatory and can be up to 31 characters in length. It must be unique. It could simply be the name of the database manager, although to maintain its uniqueness in more complicated configurations it could, for example, also include the name of the database being updated.

The name that you choose should be meaningful because the queue manager uses it to refer to this database manager instance both in messages and in output when the **dspmqtrn** command is used.

Once you have chosen a name, **do not change this attribute**. Information about changing configuration information is given in "Changing configuration information" on page 123.

SwitchFile=name
This is the fully-qualified name of the database manager's XA switch load file. This is a mandatory attribute.

XAOpenString=string
This is a string of data that is passed in calls to the database manager's xa_open entry point. The format for this string depends on the particular database manager, but it should usually identify the name of the database that is to be updated.

This is an optional attribute; if it is omitted a blank string is assumed.

XACloseString=string
This is a string of data that is passed in calls to the database manager's xa_close entry point. The format for this string depends on the particular database manager.

This is an optional attribute; if it is omitted a blank string is assumed.

ThreadOfControl=THREAD|PROCESS
The *ThreadOfControl* value can be THREAD or PROCESS. The queue manager uses it for serialization purposes.

If the database manager is "thread-safe", the value for *ThreadOfControl* can be THREAD, and the queue manager can call the database manager from multiple threads at the same time.

If the database manager is not thread-safe, the value for *ThreadOfControl* should be PROCESS. The queue manager serializes all calls to the database manager so that only one call at a time is made from within a particular process.

See "The XAResourceManager stanza" on page 169 for fuller descriptions of these attributes.

"Oracle configuration" gives more information about the specific tasks you need to perform to configure MQSeries with each of the supported database managers.

Oracle configuration

You need to perform the following tasks:

- Check Oracle level and apply patches if you have not already done so.
- Check environment variable settings.
- Enable Oracle XA support.
- Create the Oracle switch load file.
- Add XAResourceManager configuration information to the `qm.ini` file.
- Change the Oracle configuration parameters, if necessary.

Minimum supported levels for Oracle

The minimum supported level of Oracle on OpenVMS is 8.1.6.

Checking the environment variable settings

Ensure that your Oracle environment variables are set for queue manager processes as well as in your application processes. In particular, the following environment variables should always be set **prior** to starting the queue manager:

ORACLE_HOME
Is the Oracle home directory

ORACLE_SID
Is the Oracle SID being used

Enabling Oracle XA support

You need to ensure that Oracle XA support is enabled. In particular, an Oracle shared library must have been created; this happens during installation of the Oracle XA library.

During installation of Oracle8, the library is built automatically. If you need to rebuild the library, refer to the *Oracle8 Administrator's Reference* publication appropriate to your platform.

Creating the Oracle switch load file

The simplest method for creating the Oracle switch load file is to use the sample file. The source code used to create the Oracle switch load file is shown in Figure 11.

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

extern struct xa_switch_t xaosw;

struct xa_switch_t * MQENTRY MQStart(void)
{
   return(&xaosw);
}
```

Figure 11. Source code for Oracle switch load file, oraswit.c

The xa.h header file that is included is shipped with MQSeries in the same directory as oraswit.c.

Creating the Oracle switch load file on OpenVMS systems

To create the Oracle switch load file on OpenVMS systems, oraswit.c must be compiled and linked against the Oracle client library ″oraclient_v816.exe″.

1. Create the directory into which the Oracle switch load file, oraswit, will be built.
2. Copy the following files from `mqs_examples:[xatm]` into this directory:
   - xa.h
   - oraswit.c
3. Compile the copied source file (oraswit.c). For example:

```
$ cc oraswit0.c
```

4. Generate the switch load file:

```
$ link/share oraswit0.obj, sys$input/options
ora_root:[util]oraclient_v816.exe/share
SYMBOL_VECTOR=(MQStart=PROCEDURE)
```

5. The MQStart procedure is dynamically loaded at runtime from the generated image. Therefore, either a logical name must be defined in the system logical name table to point to the generated file without the `".exe"` extension or the file copied to sys$share. For example, if the location of the built file is disk$a_device:[a_directory]oraswit0.exe, then use one of the following:

```
$ define/sys/exec oraswit0 disk$a_device:[a_directory]oraswit0
```

   or

```
$ copy disk$a_device:[a_directory]oraswit0.exe sys$share:oraswit0.exe
```

Adding XAResourceManager configuration information for Oracle

The next step is to modify the `qm.ini` configuration file of the queue manager, to define Oracle as a participant in global units of work. You need to add an XAResourceManager stanza with the following attributes:

Name=name
This attribute is mandatory. Choose a suitable name for this participant. You could include the name of the database being updated.

SwitchFile=name
This attribute is mandatory. The fully-qualified name of the Oracle switch load file.

Oracle configuration

XAOpenString=string

The XA open string for Oracle has the following format:

```
Oracle_XA+Acc=P//|P/userName/passWord
         +SesTm=sessionTimeLimit
         [+DB=dataBaseName]
         [+GPwd=P/groupPassWord]
         [+LogDir=logDir]
         [+MaxCur=maximumOpenCursors]
         [+SqlNet=connectString]
```

where:

Acc= Is mandatory and is used to specify user access information. `P//` indicates that no explicit user or password information is provided and that the **ops$login** form is to be used. `P/userName/passWord` indicates a valid ORACLE user ID and the corresponding password.

SesTm=
Is mandatory and is used to specify the maximum amount of time that a transaction can be inactive before the system automatically deletes it. The unit of time is in seconds.

DB= Is used to specify the database name, where DataBaseName is the name Oracle precompilers use to identify the database. This field is required only when applications explicitly specify the database name (that is, use an AT clause in their SQL statements).

GPwd=
GPwd is used to specify the server security password, where P/groupPassWord is the server security group password name. Server security groups provide an extra level of protection for different applications running against the same ORACLE instance. The default is an ORACLE-defined server security group.

LogDir=
LogDir is used to specify the directory on a local machine where the Oracle XA library error and tracing information can be logged. If a value is not specified, the current directory is assumed. Make sure that user mqm has write-access to this directory.

MaxCur=
MaxCur is used to specify the number of cursors to be allocated when the database is opened. It serves the same purpose as the precompiler option, maxopencursors.

SqlNet=
SqlNet is used to specify the SQL*Net connect string that is used to log on to the system. The connect string can be either an SQL*Net V1 string, SQL*Net V2 string, or SQL*Net V2 alias. This field is required when you are setting up Oracle on a machine separate from the queue manager.

See the *Oracle8 Server Application Developer's Guide* (Part Number A54642-01) for more information.

XACloseString=string

Oracle does not require an XA close string.

ThreadOfControl=THREAD|PROCESS

You do not need to specify this parameter on MQSeries for Compaq OpenVMS platforms.

For fuller descriptions of each of these attributes, see "The XAResourceManager stanza" on page 169.

In Figure 12, the database to be updated is called MQBankDB. Note that it is recommended to add a `LogDir` to the XA open string so that all error and tracing information is logged to the same place. It is assumed that the Oracle switch load file was copied to the sys$share directory after it had been created.

```
XAResourceManager:
  Name=Oracle MQBankDB
  SwitchFile=sys$share:oraswit0
  XAOpenString=Oracle_XA+Acc=P/jim/tiger+SesTm=35+LogDir=/tmp/ora.log+DB=MQBankDB
```

Figure 12. Sample XAResourceManager entry for Oracle

Changing Oracle configuration parameters

The queue manager and user applications use the user ID specified in the XA open string when they connect to Oracle.

- **Database privileges** The Oracle user ID specified in the open string must have the privileges to access the DBA_PENDING_TRANSACTIONS view.

  The necessary privilege can be given using the following command, where `userID` is the user ID for which access is being given.

```
grant select on DBA_PENDING_TRANSACTIONS to userID;
```

  See "Chapter 7. Protecting MQSeries objects" on page 73 for more information about security.

Administration tasks

In normal operations only a minimal amount of administration is necessary after you have completed the configuration steps. The administration job is made easier because the queue manager is tolerant of database managers not being available. In particular this means that:

- The queue manager can be started at any time without first starting each of the database managers.
- The queue manager does not need to be stopped and restarted if one of the database managers becomes unavailable.

This allows you to start and stop the queue manager independently from the database managers, and vice versa if the database manager supports it.

Whenever contact is lost between the queue manager and a database manager they need to resynchronize when both become available again.

Resynchronization is the process by which any in-doubt units of work involving that database are completed. In general, this occurs automatically without the need for user intervention. The queue manager asks the database manager for a list of units of work in which it is in doubt. Next it instructs the database manager to either commit or rollback each of these in-doubt units of work.

When the queue manager stops, it needs to resynchronize with each database manager instance during restart. When an individual database manager becomes unavailable, only that database manager need be resynchronized the next time the queue manager notices that the database manager is available again.

The queue manager attempts to regain contact with an unavailable database manager automatically as new global units of work are started. Alternatively, the **rsvmqtrn** command can be used to resolve explicitly all in-doubt units of work.

In-doubt units of work

A database manager may be left with in-doubt units of work if contact with the queue manager is lost after the database manager has been instructed to PREPARE. Until the database manager receives the COMMIT or ROLLBACK outcome from the queue manager, it needs to retain the database locks associated with the updates.

Because these locks prevent other applications from updating, or maybe reading, database records, resynchronization needs to take place as soon as possible.

If for some reason you cannot wait for the queue manager to resynchronize with the database automatically, you could use facilities provided by the database manager to commit or rollback the database updates manually. This is called making a *heuristic* decision and should be used only as a last resort because of the possibility of compromising data integrity; you may end up committing the database updates when all of the other participants rollback, or vice versa.

It is far better to restart the queue manager, or use the **rsvmqtrn** command when the database has been restarted, to initiate automatic resynchronization.

Using the dspmqtrn command

While a database manager is unavailable it is possible to use the **dspmqtrn** command to check the state of outstanding units of work (UOWs) involving that database.

When a database manager becomes unavailable, before the two-phase commit process is entered, any in-flight UOWs in which it was participating are rolled back. The database manager itself rolls back its in-flight UOWs when it next restarts.

The **dspmqtrn** command displays only those units of work in which one or more participants are in doubt, awaiting the COMMIT or ROLLBACK from the queue manager.

For each of these units of work the state of each of the participants is displayed. If the unit of work did not update the resources of a particular resource manager, it is not displayed.

With respect to an in-doubt unit of work, a resource manager is said to have done one of the following things:

Prepared	The resource manager is prepared to commit its updates.
Committed	The resource manager has committed its updates.
Rolled-back	The resource manager has rolled back its updates.
Participated	The resource manager is a participant, but has not prepared, committed, or rolled back its updates.

Note that the queue manager does not remember the individual states of the participants when the queue manager restarts. If the queue manager is restarted, but is unable to contact a database manager, then the in-doubt units of work in which that database manager was participating are not resolved during restart. In this case, the database manager is reported as being in *prepared* state until such time as resynchronization has occurred.

Whenever the **dspmqtrn** command displays an in-doubt UOW, it first lists all the possible resource managers that could be participating. These are allocated a unique identifier, *RMId*, which is used instead of the *Name* of the resource managers when reporting their state with respect to an in-doubt UOW.

Figure 13 shows the result of issuing the following command:

```
dspmqtrn -m MY_QMGR
```

```
AMQ7107: Resource manager 0 is MQSeries.
AMQ7107: Resource manager 1 is Oracle MQBankDB
AMQ7107: Resource manager 2 is Oracle MQFeeDB

AMQ7056: Transaction number 0,1.
    XID: formatID 5067085, gtrid_length 12, bqual_length 4
         gtrid [3291A5060000201374657374]
         bqual [00000001]
AMQ7105: Resource manager 0 has committed.
AMQ7104: Resource manager 1 has prepared.
AMQ7104: Resource manager 2 has prepared.
```

Figure 13. Sample dspmqtrn output

The output from Figure 13 shows that there are three resource managers associated with the queue manager. The first is the resource manager `0`, which is the queue manager itself. The other two resource manager instances are the `MQBankDB` and `MQFeeDB` Oracle databases.

The example shows only a single in-doubt unit of work. A message is issued for all three resource managers, which means that updates had been made to the queue manager and both Oracle databases within the unit of work.

The updates made to the queue manager, resource manager **0**, have been *committed*. The updates to the Oracle databases are in *prepared* state, which means that Oracle must have become unavailable before it was called to commit the updates to the *MQBankDB* and *MQFeeDB* databases.

The in-doubt unit of work has an external identifier called an XID. This is the identifier that Oracle associates with the updates.

Using the rsvmqtrn command

The output shown in Figure 13 showed a single in-doubt UOW in which the commit decision had yet to be delivered to both Oracle databases.

In order to complete this unit of work, the queue manager and Oracle need to resynchronize when Oracle next becomes available. The queue manager uses the start of new units of work as an opportunity to attempt to regain contact with Oracle. Alternatively, you can instruct the queue manager to resynchronize explicitly using the **rsvmqtrn** command. You should do this soon after Oracle has been restarted so that any database locks associated with the in-doubt unit of work are released as quickly as possible.

This is achieved using the **-a** option which tells the queue manager to resolve all in-doubt units of work. In the following example, Oracle had been restarted so the queue manager was able to resolve the in-doubt unit of work:

```
rsvmqtrn -m MY_QMGR -a
```

Any in-doubt transactions have been resolved.

Mixed outcomes and errors

Although the queue manager uses a two-phase commit protocol this does not completely remove the possibility of some units of work completing with mixed outcomes. This is where some participants commit their updates, and some back out their updates.

Units of work that complete with a mixed outcome have serious implications because shared resources are no longer in a consistent state.

Mixed outcomes are mainly caused when heuristic decisions are made about units of work instead of allowing the queue manager to resolve in-doubt units of work itself.

Whenever the queue manager detects heuristic damage it produces FFST® information and documents the failure in its error logs, with one of two messages:

- If a database manager rolled back instead of committing:

```
AMQ7606 A transaction has been committed but one or more resource
        managers have rolled back.
```

- If a database manager committed instead of rolling back:

```
AMQ7607 A transaction has been rolled back but one or more resource
        managers have committed.
```

Further messages are issued that identify the databases that are heuristically damaged. It is then your responsibility to perform recovery steps local to the affected databases so that consistency is restored. This is a complicated procedure in which you need first to isolate the update that has been wrongly committed or rolled back, then to undo or redo the database change manually.

Damage occurring due to software errors is less likely. Units of work affected in this way have their transaction number reported by message AMQ7112. The participants may be in an inconsistent state.

```
rsvmqtrn -m MY_QMGR

AMQ7107: Resource manager 0 is MQSeries.
AMQ7107: Resource manager 1 is Oracle MQBankDB
AMQ7107: Resource manager 2 is Oracle MQFeeDB

AMQ7112: Transaction number 0,1 has encountered an error.
    XID: formatID 5067085, gtrid_length 12, bqual_length 4
         gtrid [3291A5060000201374657374]
         bqual [00000001]
AMQ7105: Resource manager 0 has committed.
AMQ7104: Resource manager 1 has prepared.
AMQ7104: Resource manager 2 has rolled back.
```

Figure 14. Sample dspmqtrn output for a transaction in error

The queue manager does not attempt to recover from such failures until the next queue manager restart. In Figure 14, this would mean that the updates to resource manager **1**, the MQBankDB database, would be left in *prepared* state even if the **rsvmqtrn** was issued to resolve the unit of work.

Changing configuration information

After the queue manager has successfully started to coordinate global units of work you should be wary about making changes to any of the XAResourceManager stanzas in the `qm.ini` file.

If you do need to change the `qm.ini` file you can do so at any time, but the changes do not take effect until after the queue manager has been restarted. For example, if you need to alter the XA open string passed to a database manager, you need to restart the queue manager for your change to take effect.

Note that if you remove an XAResourceManager stanza you are effectively removing the ability for the queue manager to contact that database manager.

You should ***never*** change the *Name* attribute in any of your XAResourceManager stanzas. This attribute uniquely identifies that database manager instance to the queue manager. If this unique identifier is changed, the queue manager assumes that the database manager instance has been removed and a completely new instance has been added. The queue manager still associates outstanding units of work with the old *Name*, possibly leaving the database in an in-doubt state.

Removing database manager instances

If you do need to remove a database or database manager from your configuration permanently, you should first ensure that the database is not in doubt. You should perform this check before you restart the queue manager. Most database managers provide commands for listing in-doubt transactions. If there are any in-doubt transactions, first allow the queue manager to resynchronize with the database manager before you remove its XAResourceManager stanza.

If you fail to observe this procedure the queue manager still remembers all in-doubt units of work involving that database. A warning message, AMQ7623, is issued every time the queue manager is restarted. If you are never going to configure this database with the queue manager again, you can instruct the queue manager to forget about the participation of the database in its in-doubt transactions using the -r option of the **rsvmqtrn** command.

Note: The queue manager will finally forget about transactions only when syncpoint processing has been completed with all participants.

There are times when you might need to remove an XAResourceManager stanza temporarily. This is best achieved by commenting out the stanza so that it can be easily reinstated at a later time. You may decide to take this action if you are suffering errors every time the queue manager contacts a particular database or database manager. Temporarily removing the XAResourceManager entry concerned allows the queue manager to start global units of work involving all of the other participants. An example of a commented out *XAResourceManager* stanza follows:

```
# This database has been temporarily removed
#XAResourceManager:
#  Name=Oracle MQBankDB
#  SwitchFile=sys$share:oraswit0
#  XAOpenString=MQBankDB
```

Figure 15. Commented out XAResourceManager stanza

Chapter 11. Recovery and restart

A messaging system ensures that messages entered into the system are delivered to their destination. This means that it must provide a method of tracking the messages in the system, and of recovering messages if the system fails for any reason.

MQSeries ensures that messages are not lost by maintaining records (logs) of the activities of the queue managers that handle the receipt, transmission, and delivery of messages. It uses these logs for three types of recovery:

1. *Restart recovery*, when you stop MQSeries in a planned way.
2. *Crash recovery*, when MQSeries is stopped by an unexpected failure.
3. *Media recovery*, to restore damaged objects.

In all cases, the recovery restores the queue manager to the state it was in when the queue manager stopped. Any in-flight transactions are rolled back, removing from the queues any messages that were not committed at the time the queue manager stopped. Recovery restores all persistent messages; non-persistent messages are lost during the process.

The rest of this chapter introduces the concepts of recovery and restart in more detail and then tells you how to recover if problems occur. It covers the following topics:

- “Making sure that messages are not lost (logging)”
- “Checkpointing – ensuring complete recovery” on page 127
- “Calculating size of log” on page 130
- “Managing logs” on page 131
- “Using the log for recovery” on page 133
- “Protecting MQSeries log files” on page 135
- “Backup and restore” on page 136
- “Recovery scenarios” on page 137
- “Dumping the contents of the log using the dmpmqlog command” on page 138.

Making sure that messages are not lost (logging)

MQSeries records all significant changes to the data controlled by the queue manager in a log. This includes the creation and deletion of objects, all persistent message updates, transaction states, changes to object attributes, and channel activities. Therefore, the log contains the information you need to recover all updates to message queues by:

- Keeping records of queue manager changes.
- Keeping records of queue updates for use by the restart process.
- Enabling you to restore data after a hardware or software failure.

What logs look like

An MQSeries log consists of two components:

1. One or more files of log data
2. A log control file

There are a number of log files which contain the data being recorded. You can define the number and size (as explained in “Calculating size of log” on page 130), or take the system default of 3 files, each 4MB in size.

Logging

When you create a queue manager, the number of log files you define is the number of *primary* log files allocated. If you do not specify a number, the default value is used. If you have not changed the log path, they are created in the directory:

```
MQS_ROOT:[MQM.LOG.QmName.ACTIVE]
```

MQSeries starts with these primary log files, but, if the log starts to get full, allocates *secondary* log files. It does this dynamically, and removes them when the demand for log space reduces. By default, up to two secondary log files can be allocated, providing a further 8MB of disk space. The default number can also be changed, see "Chapter 13. Configuring MQSeries" on page 159.

Log control file

The log control file contains the information needed to monitor the use of log files: their size and location, the name of the next available file, and so on.

Note: You should ensure that the logs created when you start a queue manager are large enough to accommodate the size and volume of messages that your applications will handle. The default log numbers and sizes will require modification to meet your requirements. How to change the default values is described on page 130.

Types of logging

In MQSeries, the number of files that are used for logging depends on the file size, the number of messages you have received, and the length of the messages. There are two ways of maintaining records of queue manager activities: circular logging and linear logging.

Circular logging

Use circular logging if all you want is restart recovery, using the log to roll back transactions that were in progress when the system stopped.

Circular logging keeps all restart data in a ring of log files. Logging fills the first file in the ring, then moves on to the next, and so on, until all the files are filled. It then goes back to the first file in the ring and starts again. This continues as long as the product is in use and has the advantage that you never run out of log files.

The above is a simple explanation of circular logging. However, there is a complication. The log entries required to restart the queue manager without loss of data are kept until they are no longer required to ensure queue manager data recovery. The mechanism for releasing log files for reuse is described in "Checkpointing – ensuring complete recovery" on page 127. For now, you should know that MQSeries uses secondary log files to extend the log capacity as necessary.

Linear logging

Use linear logging if you want both restart recovery and media or forward recovery (recreating lost or damaged data by replaying the contents of the log).

Linear logging keeps the log data in a continuous sequence of files. Space is not reused, so you can always retrieve any record logged from the time that the queue manager was created.

As disk space is finite, you may have to think about some form of archiving. It is an administrative task to manage your disk space for the log, reusing or extending the existing space as necessary.

The number of log files used with linear logging can be very large depending on your message flow and the age of your queue manager. However, there are a number of files which are said to be active. Active files contain the log entries required to restart the queue manager. The number of active log files is usually the same as the number of primary log files as defined in the configuration files. (See "Calculating size of log" on page 130 for further details of how to define the number.)

The key event that controls whether a log file is termed active or not is a *checkpoint*. An MQSeries checkpoint is a group of log records containing information to allow a successful restart of the queue manager. Any information recorded previously is not required to restart the queue manager and can therefore be termed inactive. (See "Checkpointing – ensuring complete recovery" for further information about checkpointing.)

You must decide when inactive log files are no longer required. You may select to archive them, or you may delete them as being no longer of interest to your operation. Refer to "Managing logs" on page 131 for further information about the disposition of log files.

If a new checkpoint is recorded in the second, or later, primary log file, then the first file becomes inactive and a new primary file is formatted and added to the end of the primary pool, restoring the number of primary files available for logging. In this way the primary log file pool can be seen to be a current set of files in an ever extending list of log files. Again, it is an administrative task to manage the inactive files according to the requirements of your operation.

Although secondary log files are defined for linear logging, they are not used in normal operation. If a situation should arise when, probably due to long-lived transactions, it is not possible to free a file from the active pool because it may still be required for a restart, secondary files are formatted and added to the active log file pool.

If the number of secondary files available is used up, requests for most further operations requiring log activity will be refused with an MQRC_RESOURCE_PROBLEM being returned to the application.

Both types of logging can cope with unexpected loss of power assuming that there is no hardware failure.

Checkpointing – ensuring complete recovery

Persistent updates to message queues happen in two stages. First, the records representing the update are written to the log, then the queue file is updated. The log files can thus become more up-to-date than the queue files. To ensure that restart processing begins from a consistent point, MQSeries uses checkpoints. A checkpoint is a point in time when the record described in the log is the same as the record in the queue. The checkpoint itself consists of the series of log records needed to restart the queue manager; for example, the state of all transactions (that is, units of work) active at the time of the checkpoint.

Checkpoints are generated automatically by MQSeries. They are taken when the queue manager starts, at shutdown, when logging space is running low, and after every 1000 operations logged. As the queues handle further messages, the checkpoint record becomes inconsistent with the current state of the queues.

Checkpointing

When MQSeries is restarted, it locates the latest checkpoint record in the log. This information is held in the checkpoint file that is updated at the end of every checkpoint. The checkpoint record represents the most recent point of consistency between the log and the data. The data from this checkpoint is used to rebuild the queues as they existed at the checkpoint time. When the queues are recreated, the log is then played forward to bring the queues back to the state they were in before system failure or close down.

MQSeries maintains internal pointers to the head and tail of the log. It moves the head pointer to the most recent checkpoint that is consistent with recovering message data.

Checkpoints are used to make recovery more efficient, and to control the reuse of primary and secondary log files.

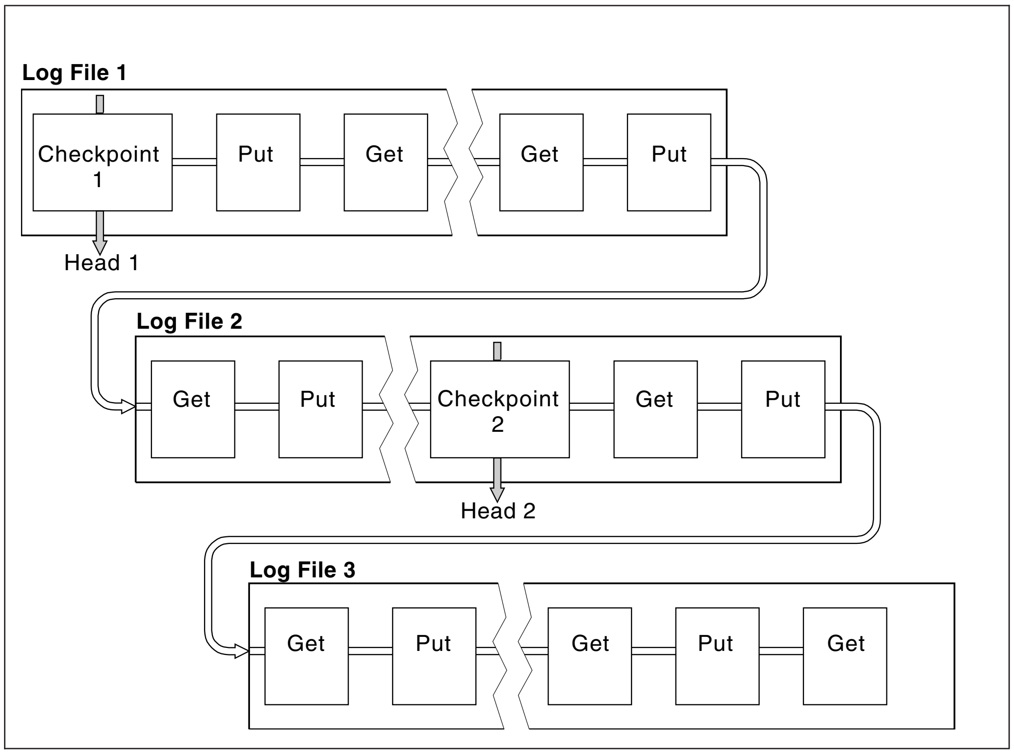


Figure 16. Checkpointing. For simplicity, only the ends of the log files are shown.

In Figure 16, all records before the latest checkpoint, checkpoint 2, are no longer needed by MQSeries. The queues can be recovered from the checkpoint information and any later log entries. For circular logging, any freed files prior to the checkpoint can be reused. For a linear log, the freed log files no longer need to be accessed for normal operation and become inactive. In the example, the queue head pointer is moved to point at the latest checkpoint, Checkpoint 2, which then becomes the new queue head, head 2. Log File 1 can now be reused.

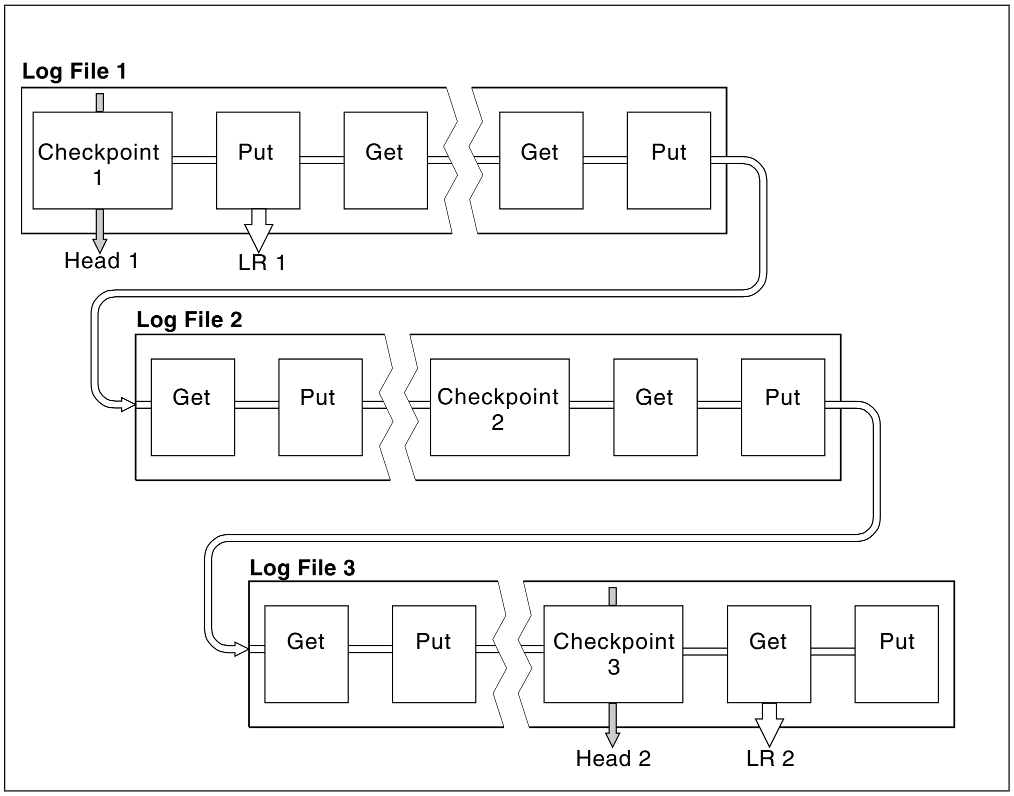


Figure 17. Checkpointing with a long-running transaction. For simplicity, only the ends of the log files are shown.

Figure 17 shows how a long-running transaction affects reuse of log files. In the example, a long-running transaction has caused an entry to the log, shown as LR 1, after the first checkpoint shown. The transaction does not complete, shown as LR 2, until after the third checkpoint. All the log information from LR 1 onwards is retained to allow recovery of that transaction, if necessary, until it has completed.

After the long-running transaction has completed, at LR 2, the head of the log is moved to checkpoint 3, the latest logged checkpoint. The files containing log records prior to checkpoint 3, Head 2, are no longer needed. If you are using circular logging, the space can be reused.

If the primary log files are completely filled before the long-running transaction completes, secondary log files are used to avoid the risk of a log full situation if possible.

When the log head is moved and you are using circular logging, the primary log files may become eligible for reuse and the logger, after filling the current file, reuses the first primary file available to it. If instead you are using linear logging, the log head is still moved down the active pool and the first file becomes inactive. A new primary file is formatted and added to the bottom of the pool in readiness for future logging activities.

Calculating size of log

After deciding whether the queue manager should use circular or linear logging, your next task is to estimate the size of the log that the queue manager will need. The size of the log is determined by the following log configuration:

LogFilePages
The size of each primary and secondary log file in units of 4 KB pages

LogPrimaryFiles
The number of preallocated primary log files

LogSecondaryFiles
The number of secondary log files that can be created for use when the primary log files are full

Table 6 shows the amount of data the queue manager logs for various operations. Most operations performed by the queue manager require a minimal amount of log space, however, when a persistent message is put to a queue, all of the message data must be written to the log to make recovery of the message possible. Therefore, the size of the log depends, typically, upon the number and size of the persistent messages the queue manager needs to handle.

Table 6. Log overhead sizes (All values are approximate)

Operation	Size
Put persistent message	750 bytes + message length. If the message is large, it is divided into segments of 15700 bytes, each with a 300–byte overhead.
Get message	260 bytes
Syncpoint, commit	750 bytes
Syncpoint, roll-back	1000 bytes + 12 bytes for each get or put to be rolled back
Create object	1500 bytes
Delete object	300 bytes
Alter attributes	1024 bytes
Record media image	800 bytes + image. The image is divided into segments of 15700 bytes, each having a 300–byte overhead
Checkpoint	750 bytes + 200 bytes for each active unit of work. Additional data may be logged for any uncommitted puts or gets that have been buffered for performance reasons.

Notes:

1. The number of primary and secondary log files can be changed each time the queue manager is started.
2. The log file size cannot be changed and needs to be determined **before** the queue manager is created.
3. The number of primary log files and the log file size determine the amount of the log space that is preallocated when the queue manager is created. You are advised to organize this space as a smaller number of larger log files rather than a larger number of small log files.
4. The total number of primary and secondary log files cannot exceed 63, which, in the presence of long-running transactions, limits the maximum amount of

log space that can be made available to the queue manager for restart recovery. The amount of log space the queue manager may need to use for media recovery does not share this limit.

5. When *circular* logging is being used, the queue manager reuses primary log space. This means that the queue manager's log can be smaller than the amount of data you have estimated that the queue manager needs to log. The queue manager will, up to a limit, allocate a secondary log file when a log file becomes full, and the next primary log file in the sequence is not available.
6. Primary log files are made available for reuse during checkpoint. The queue manager takes both the primary and secondary log space into consideration before a checkpoint is taken because the amount of log space is running low.

   If you do not define more primary log files than secondary log files, the queue manager may allocate secondary log files before a checkpoint is taken. This makes the primary log files available for reuse.

Managing logs

Over time, some of the log records written become unnecessary for restarting the queue manager, and the queue manager reclaims freed space in the log files. This activity is transparent to the user and you do not usually see the amount of disk space used reduce because the space allocated is quickly reused.

Of the log records, only those written since the start of the last complete checkpoint, and those written by any active transactions, are needed to restart the queue manager. Thus, the log may fill if a checkpoint has not been taken for a long time, or if a long-running transaction wrote a log record a long time ago. The queue manager tries to take checkpoints sufficiently frequently to avoid the first problem.

When a long-running transaction fills the log, attempts to write log records fail and some MQI calls return MQRC_RESOURCE_PROBLEM. (Space is reserved to commit or rollback all in-flight transactions, so MQCMIT or MQBACK should not fail.)

The queue manager rolls back transactions that consume too much log space. An application whose transaction is rolled back in this way is unable to perform subsequent MQPUT or MQGET operations specifying syncpoint under the same transaction. An attempt to put or get a message under syncpoint in this state returns MQRC_BACKED_OUT. The application may then issue MQCMIT, which returns MQRC_BACKED_OUT, or MQBACK and start a new transaction. When the transaction consuming too much log space has been rolled back, its log space is released and the queue manager continues to operate normally.

If the log fills, a message is issued (AMQ7463). In addition, if the log fills because a long-running transaction has prevented the space being released, message AMQ7465 is issued.

Finally, if records are being written to the log faster than the asynchronous housekeeping processes can handle them, message AMQ7466 is issued. If you see this message, you should increase the number of log files or reduce the amount of data being processed by the queue manager.

What happens when a disk gets full

The queue manager logging component can cope with a full disk, and with full log files. If the disk containing the log fills, the queue manager issues message AMQ6708 and an error record is taken.

The log files are created at their maximum size, rather than being extended as log records are written to them. This means that MQSeries can only run out of disk space when it is creating a new file. It therefore cannot run out of space when it is writing a record to the log. MQSeries always knows how much space is available in the existing log files and manages the space within the files accordingly.

If you fill the drive containing the log files, you may be able to free some disk space. If you are using a linear log, there may be some inactive log files in the log directory which you can copy to another drive or device. If you still run out of space, check that the configuration of the log in the queue manager configuration file is correct. You may be able to reduce the number of primary or secondary log files so that the log does not outgrow the available space. Note that it is not possible to alter the size of the log files for an existing queue manager. The queue manager assumes that all log files are the same size.

Managing log files

If you are using circular logging, ensure that there is sufficient space to hold the log files. You do this when you configure your system (see “The LogDefaults stanza” on page 163 and “The Log stanza” on page 167). The amount of disk space used by the log, including the space required for secondary files to be created when required, is limited by the configured size of the disk.

If you are using a linear log, the log files are added continually as data is logged, and the amount of disk space used increases with time. If the rate of data being logged is high, disk space is consumed rapidly by new log files.

Over time, the older log files for a linear log are no longer required to restart the queue manager or perform media recovery of any damaged objects. Periodically, the queue manager issues a pair of messages to indicate which of the log files is required:

- Message AMQ7467 gives the name of the oldest log file needed to restart the queue manager. This log file and all newer log files must be available during queue manager restart.
- Message AMQ7468 gives the name of the oldest log file needed to do media recovery.

Any log files older than these do not need to be online. You can copy them to an archive medium such as tape for disaster recovery, and remove them from the active log directory. Any log files needed for media recovery but not for restart can also be off-loaded to an archive.

If any log file that is needed cannot be found, operator message AMQ6767 is issued. Make the log file, and all subsequent log files, available to the queue manager and retry the operation.

Note: When performing media recovery, all the required log files must be available in the log file directory at the same time. Make sure that you take regular media images of any objects you may wish to recover to avoid running out of disk space to hold all the required log files.

Log file location

When choosing a location for your log files, remember that operation is severely impacted if MQSeries fails to format a new log because of lack of disk space.

If you are using a circular log, ensure that there is sufficient space on the drive for at least the configured primary log files. You should also leave space for at least one secondary log file which is needed if the log has to grow.

If you are using a linear log, you should allow considerably more space; the space consumed by the log increases continuously as data is logged.

Ideally, the log files should be placed on a separate disk drive from the queue manager data. This has benefits in terms of performance. It may also be possible to place the log files on multiple disk drives in a mirrored arrangement. This gives protection against failure of the drive containing the log. Without mirroring, you could be forced to go back to the last backup of your MQSeries system.

Using the log for recovery

There are several ways that your data can be damaged. MQSeries for Compaq OpenVMS helps you recover from:
- A damaged data object
- A power loss in the system
- A communications failure
- A damaged log volume

This section looks at how the logs are used to recover from these problems.

Recovering from problems

MQSeries can recover from both communications failures and loss of power. In addition, it is sometimes possible to recover from other types of problem, such as inadvertent deletion of a file.

In the case of a communications failure, messages remain on queues until they are removed by a receiving application. If the message is being transmitted, it remains on the transmission queue until it can be successfully transmitted. To recover from a communications failure, it is normally sufficient simply to restart the channels using the link that failed.

If you lose power, when the queue manager is restarted MQSeries restores the queues to their state at the time of the failure. This ensures that no persistent messages are lost. Nonpersistent messages are discarded; they do not survive when MQSeries stops.

There are ways in which an MQSeries object can become unusable, for example due to inadvertent damage. You then have to recover either your complete system or some part of it. The action required depends on when the damage is detected, whether the log method selected supports media recovery, and which objects are damaged.

Media recovery

Media recovery means recreating objects from information recorded only in a linear log. Media recovery is not supported with circular logging. For example, if an object file is inadvertently deleted, or becomes unusable for some other reason, media recovery can be used to recreate it. The information in the log required for media recovery of an object is called a *media image*. Media images can be recorded manually, using the **rcdmqimg** command, or automatically in certain circumstances.

A media image is a sequence of log records containing an image of an object from which the object itself can be recreated.

The first log record required to recreate an object is known as its *media recovery record*; it is the start of the latest media image for the object. The media recovery record of each object is one of the pieces of information recorded during a checkpoint.

When recreating an object from its media image, it is also necessary to replay any log records describing updates performed on the object since the last image was taken.

Consider, for example, a local queue that has an image of the queue object taken before a persistent message is put onto the queue. In order to recreate the latest image of the object, it is necessary to replay the log entries recording the putting of the message to the queue, as well as replaying the image itself.

When an object is created, the log records written contain enough information to completely recreate the object. These records make up the object's first media image. Subsequently, media images are recorded automatically by the queue manager when:

- Images of all process objects and non-local queues are taken at each shutdown.
- Local queue images are taken when the queue becomes empty.

Media images can also be recorded manually using the **rcdmqimg** command, described in "rcdmqimg (Record media image)" on page 262.

Recovering media images

MQSeries automatically recovers some objects from their media image if it finds that they are corrupt or damaged. In particular, this applies to objects found to be damaged during the normal queue manager startup. If any transaction was incomplete at the time of the last shutdown of the queue manager, any queue affected is also recovered automatically in order to complete the startup operation.

You must recover other objects manually, using the **rcrmqobj** command. This command replays the records in the log to recreate the MQSeries object. The object is recreated from its latest image found in the log, together with all applicable log events between the time the image was saved and the time the recreate command is issued. Should an MQSeries object become damaged, the only valid actions that can be performed are either to delete it or to recreate it by this method. Note, however, that nonpersistent messages cannot be recovered in this way.

See "rcrmqobj (Recreate object)" on page 264 for further details of the **rcrmqobj** command.

It is important to remember that you must have the log file containing the media recovery record, and all subsequent log files, available in the log file directory when attempting media recovery of an object. If a required file cannot be found, operator message AMQ6767 is issued and the media recovery operation fails. If you do not take regular media images of the objects that you may wish to recreate, you can get into the situation where you have insufficient disk space to hold all the log files required to recreate an object.

Recovering damaged objects during startup

If the queue manager discovers a damaged object during startup, the action it takes depends on the type of object and whether the queue manager is configured to support media recovery.

If the queue manager object is damaged, the queue manager cannot start unless it can recover the object. If the queue manager is configured with a linear log, and thus supports media recovery, MQSeries automatically tries to recreate the MQSeries object from its media images. If the log method selected does not support media recovery, you can either restore a backup of the queue manager or delete the queue manager.

If any transactions were active when the queue manager stopped, the local queues containing the persistent, uncommitted messages put or got inside these transactions are also needed to start the queue manager successfully. If any of these local queues are found to be damaged, and the queue manager supports media recovery, it automatically attempts to recreate them from their media images. If any of the queues cannot be recovered, MQSeries cannot start.

If any damaged local queues containing uncommitted messages are discovered during startup processing on a queue manager that does not support media recovery, the queues are marked as damaged objects and the uncommitted messages on them are ignored. This is because it is not possible to perform media recovery of damaged objects on such a queue manager and the only action left is to delete them. Message AMQ7472 is issued to report any damage.

Recovering damaged objects at other times

Media recovery of objects is only automatic during startup. At other times, when object damage is detected, operator message AMQ7472 is issued and most operations using the object fail. If the queue manager object is damaged at any time after the queue manager has started, the queue manager performs a preemptive shutdown. When an object has been damaged you may delete it or, if the queue manager is using a linear log, attempt to recover it from its media image using the **rcrmqobj** command (see "rcrmqobj (Recreate object)" on page 264 for further details).

Protecting MQSeries log files

It is important that when an MQSeries queue manager is running you do not remove the log files manually. If a user inadvertently (or maliciously) deletes the log files which a queue manager needs to restart, MQSeries does not issue any errors and continues to process data including persistent messages. The queue manager shuts down normally, but will fail to restart. Media recovery of messages then becomes impossible.

Any user with the authority to remove logs that are being used by an active queue manager also has authority to delete other important queue manager resources (such as authorization files, queue files, the object catalog, and MQSeries executables). They therefore can damage, perhaps through inexperience or even intent, a running or dormant queue manager in a way against which MQSeries cannot protect itself.

Exercise caution when granting users elevated privileges or the MQM rights identifier.

Backup and restore

Periodically, you may want to take a backup of your queue manager data to provide protection against possible corruption due to hardware failures. However, because message data is often short-lived, you may choose not to take backups.

Backing up MQSeries

To take a backup of a queue manager's data, you must:

1. Ensure that the queue manager is not running.

   If your queue manager is running, stop it with the **endmqm** command.

   Note: If you try to take a backup of a running queue manager, the backup may not be consistent due to updates in progress when the files were copied.
2. Locate the directories under which the queue manager places its data and its log files.

   You can use the information in the configuration files to determine these directories. For more information about this, see "Chapter 13. Configuring MQSeries" on page 159.

   Note: You may have some difficulty in understanding the names that appear in the directory. This is because the names are transformed to ensure that they are compatible with the platform on which you are using MQSeries. For more information about name transformations, see "Understanding MQSeries file names" on page 19.
3. Take copies of all the queue manager's data and log file directories, including all subdirectories.

   Make sure that you do not miss any of the files, especially the log control file and the configuration files. Some of the directories may be empty, but they will all be required if you restore the backup at a later date, so it is advisable to save them too.
4. Ensure that you preserve the ownerships of the files. You can do this with the BACKUP command and the /BY_OWNER parameter.

Restoring MQSeries

To restore a backup of a queue manager's data, you must:

1. Ensure that the queue manager is not running.
2. Locate the directories under which the queue manager places its data and its log files. This information is held in the configuration file.
3. Clear out the directories into which you are going to place the backed up data.
4. Copy the backed up queue manager data and log files into the correct places.

Check the resulting directory structure to ensure that you have all of the required directories.

See "Appendix C. Directory structure" on page 299 for more information about MQSeries directories and subdirectories.

Make sure that you have a log control file as well as the log files. Also check that the MQSeries and queue manager configuration files are consistent so that MQSeries can look in the correct places for the restored data.

If the data was backed up and restored correctly, the queue manager will now start.

Note: Even though the queue manager data and log files are held in different directories, you should back up and restore the directories at the same time. If the queue manager data and log files have different ages, the queue manager is not in a valid state and will probably not start. If it does start, your data will almost certainly be corrupt.

Recovery scenarios

This section looks at a number of possible problems and indicates how to recover from them.

Disk drive failures

You may suffer problems with a disk drive containing either the queue manager data, the log, or both. Problems can include data loss or corruption. The three cases differ only in the part of the data that survives, if any.

In ***all*** cases you must first check the directory structure for any damage and, if necessary, repair such damage. If you lose queue manager data, there is a danger that the queue manager directory structure has been damaged. If so, you must recreate the directory tree manually before you try to restart the queue manager. Having checked for structural damage, there are a number of alternative things you can do, depending on the type of logging that you use.

- **Where there is major damage to the directory structure or any damage to the log**, remove all the old files back to the QMgrName level, including the configuration files, the log, and the queue manager directory, restore the last backup, and try to restart the queue manager.
- **For linear logging with media recovery**, ensure the directory structure is intact and try to restart the queue manager. If the queue manager does not restart, restore a backup. If the queue manager restarts, check whether any other objects have been damaged using MQSC. Recover the ones you find, using the **rcrmqobj** command, for example:

  ```
  rcrmqobj -m QMgrName -t * *
  ```

  where QMgrName is the queue manager being recovered. -t * * indicates that any object of any type will be recovered. If only one or two objects have been reported as damaged, you may want to specify those objects by name and type here.

  Note: These commands do not apply to channels.
- **For linear logging with media recovery and with an undamaged log**, you may be able to restore a backup of the queue manager data leaving the existing log files and log control file unchanged. Starting the queue manager applies the changes from the log to bring the queue manager back to its state when the failure occurred.

  This method relies on two facts. Firstly, it is vital that the checkpoint file be restored as part of the queue manager data. This file contains the information determining how much of the data in the log must be applied to give a consistent queue manager.

Secondly, you must have the oldest log file which was required to start the queue manager at the time of the backup, and all subsequent log files, available in the log file directory.

If this is not possible, you must restore a backup of both the queue manager data and the log, both of which were taken at the same time.

- **For circular logging, or linear logging without media recovery**, you must restore the queue manager from the latest backup that you have. Once you have restored the backup, restart the queue manager and check as above for damaged objects. However, because you do not have media recovery, you must find other ways of recreating the damaged objects.

Damaged queue manager object

If the queue manager object has been reported as damaged during normal operation, the queue manager performs a preemptive shutdown. There are two ways of recovering in these circumstances depending on the type of logging you use:

- **For linear logging only**, manually delete the file containing the damaged object and restart the queue manager. Media recovery of the damaged object is automatic.
- **For circular or linear logging**, restore the last backup of the queue manager data and log and restart the queue manager.

Damaged single object

If a single object is reported as damaged during normal operation, there are two ways of recovering, depending on the type of logging you use:

- **For linear logging**, recreate the object from its media image.
- **For circular logging**, restore the last backup of the queue manager data and log and restart the queue manager.

Automatic media recovery failure

If a local queue required for queue manager startup with a linear log is damaged, and the automatic media recovery fails, restore the last backup of the queue manager data and log and restart the queue manager.

Dumping the contents of the log using the dmpmqlog command

The **dmpmqlog** command can be used to dump the contents of the queue manager log. By default all active log records are dumped, that is, the command starts dumping from the head of the log. Normally this is from the start of the last completed checkpoint.

The log can be dumped only when the queue manager is not running. Because the queue manager takes a checkpoint during shutdown, the active portion of the log usually contains a small number of log records. However, the **dmpmqlog** command can be instructed to dump more log records using one of the following options to change the start position of the dump:

- The simplest option is to start dumping from the *base* of the log. The base of the log is the first log record in the log file that contains the head of the log. The amount of additional data dumped in this case depends upon where the head of the log is positioned in the log file. If it is near to the start of the log file only a small amount of additional data is dumped. If the head is near to the end of the log file then significantly more data is dumped.

- Another option enables the start position of the dump to be specified as an individual log record. Each log record is identified by a unique *log sequence number (LSN)*. In the case of circular logging, this starting log record cannot be prior to the base of the log; this restriction does not apply to linear logs. Inactive log files may need to be reinstated before running the command. For this option a valid LSN must be specified as the start position. This must be taken from previous **dmpmqlog** output.

  For example, with linear logging you could specify the `nextlsn` from your last **dmpmqlog** output. The Next LSN appears in `Log File Header` and indicates the LSN of the next log record to be written. This can therefore be used as a start position to format all log records that have been written since the last time the log was dumped.
- The third option is for linear logs only. The dumper can be instructed to start formatting log records from any given log file extent. In this case the log dumper expects to find this log file, and each successive one, in the same directory as the active log files. This option does not apply to circular logs, because in this case the log dumper cannot access log records prior to the base of the log.

The output from the **dmpmqlog** command is the `Log File Header` and a series of formatted log records. The queue manager uses several log records to record changes to its data.

Some of the information that is formatted is of use only internally. The following list includes the most useful log records:

Log File Header
Each log has a single log file header, which is always the first thing formatted by the **dmpmqlog** command. It contains the following fields:

logactive	The number of primary log extents.
loginactive	The number of secondary log extents.
logsize	The number of 4 KB pages per extent.
baselsn	The first LSN in the log extent containing the head of the log.
nextlsn	The LSN of next log record to be written.
headlsn	The LSN of the log record at the head of the log.
tailsn	The LSN identifying the tail position of the log.
hflag1	Identifies whether log is CIRCULAR or LOG RETAIN (linear).
HeadExtentID	The log extent containing the head of the log.

Log Record Header
Each log record within the log has a fixed header containing the following information:

LSN	The log sequence number.
LogRecdType	The type of the log record.
XTranid	The transaction identifier associated with this log record (if any). A *TranType* of MQI indicates an MQ-only transaction. A *TranType* of XA is involved with other resource managers. Updates involved within the same unit of work have the same *XTranid*.

QueueName	The queue associated with this log record (if any).
Qid	The unique internal identifier for the queue.
PrevLSN	LSN of previous log record within the same transaction (if any).

Start Queue Manager
This logs that the queue manager has been started.

StartDate	The date that the queue manager was started.
StartTime	The time that the queue manager was started.

Stop Queue Manager
This logs that the queue manager has been stopped.

StopDate	The date that the queue manager was stopped.
StopTime	The time that the queue manager was stopped.
ForceFlag	The type of shutdown that was used.

Start Checkpoint
This denotes the start of a queue manager checkpoint.

End Checkpoint
This denotes the end of a queue manager checkpoint.

ChkPtLSN	The LSN of the log record that started this checkpoint.

Put Message
This logs a persistent message put to a queue. If the message was put under syncpoint, then the log record header contains a nonnull *XTranid*. The remainder of the record contains:

SpcIndex	An identifier for the message on the queue. It can be used to match the corresponding MQGET that was used to get this message from the queue. In this case a subsequent *Get Message* log record can be found containing the same QueueName and SpcIndex. At this point the SpcIndex identifier can be reused for a subsequent put message to that queue.
Data	Contained in the hex dump for this log record is various internal data followed by the Message Descriptor (eyecatcher MD) and the message data itself.

Put Part
Persistent messages that are too large for a single log record are logged as a single *Put Message* record followed by multiple *Put Part* log records.

Data	Continues the message data where the previous log record left off.

Get Message
Only gets of persistent messages are logged. If the message was got under syncpoint then the log record header contains a nonnull *XTranid*. The remainder of the record contains:

SpcIndex	Identifies the message that was got from the queue. The most recent *Put Message* log record containing the same *QueueName* and *SpcIndex* identifies the message that was got.
QPriority	The priority of the message got from the queue.

Start Transaction
Indicates the start of a new transaction. A TranType of MQI indicates an

MQ-only transaction. A TranType of XA indicates one that involves other resource managers. All updates made by this transaction will have the same *XTranid*.

Prepare Transaction
Indicates that the queue manager is prepared to commit the updates associated with the specified *XTranid*. This log record is written as part of a two-phase commit involving other resource managers.

Commit Transaction
Indicates that the queue manager has committed all updates made by a transaction.

Rollback Transaction
This log record denotes the queue manager's intention to roll back a transaction.

End Transaction
This log record denotes the end of a rolled-back transaction.

Transaction Table
This record is written during syncpoint. It records the state of each transaction that has made persistent updates. For each transaction the following information is recorded:

XTranid	Transaction identifier.
FirstLSN	LSN of first log record associated with transaction.
LastLSN	LSN of last log record associated with transaction.

Transaction Participants
This log record is written by the XA Transaction Manager component of the queue manager. It records the external resource managers that are participating in transactions. For each participant the following is recorded:

RMName	The name of the resource manager.
RMId	Resource manager identifier. This is also logged in subsequent *Transaction Prepared* log records which record global transactions in which the resource manager is participating.
SwitchFile	The switch load file for this resource manager.
XAOpenString	The XA open string for this resource manager.
XACloseString	The XA open string for this resource manager.

Transaction Prepared
This log record is written by the XA Transaction Manager component of the queue manager. It indicates that the specified global transaction has been successfully prepared. Each of the participating resource managers will be instructed to commit. The *RMId* of each prepared resource manager is recorded in the log record. If the queue manager itself is participating in the transaction a *Participant Entry* with an *RMID* of zero will be present.

Transaction Forget
This log record is written by the XA Transaction Manager component of the queue manager. It follows the *Transaction Prepared* log record when the commit decision has been delivered to each participant.

Purge Queue
This logs the fact that all messages on a queue have been purged, for example, using the RUNMQSC CLEAR command.

Queue Attributes
This logs the initialization or change of the attributes of a queue.

Create Object
Logs the creation of an MQSeries object.

ObjName	The name of the object that was created.
UserId	The user ID performing the creation.

Delete Object
Logs the deletion of an MQSeries object.

ObjName	The name of the object that was deleted.

Figure 18 on page 143 shows example output from a **dmpmqlog** command. The dump, which started at the LSN of a specific log record, was produced using the following command:

```
dmpmqlog -m "testqm" -s 0:0:0:44162
```

```
AMQ7701: DMPMQLOG command is starting.
LOG FILE HEADER
***************

counter1  . . . : 23                counter2  . . . : 23
FormatVersion . : 2                 logtype . . . . : 10
logactive . . . : 3                 loginactive . . : 2
logsize . . . . : 1024      pages
baselsn . . . . : <0:0:0:0>
nextlsn . . . . : <0:0:0:60864>
lowtranlsn  . . : <0:0:0:0>
minbufflsn  . . : <0:0:0:58120>
headlsn . . . . : <0:0:0:58120>
taillsn . . . . : <0:0:0:60863>
logfilepath . . : ""
hflag1  . . . . : 1
               -> CONSISTENT
               -> CIRCULAR
HeadExtentID  . : 1                 LastEID . . . . : 846249092
LogId . . . . . : 846249061         LastCommit  . . : 0
FirstArchNum  . : 4294967295        LastArchNum . . : 4294967295
nextArcFile . . : 4294967295        firstRecFile  . : 4294967295
firstDlteFile . : 4294967295        lastDeleteFile  : 4294967295
RecHeadFile . . : 4294967295        FileCount . . . : 3
frec_trunclsn . : <0:0:0:0>
frec_readlsn  . : <0:0:0:0>
frec_extnum . . : 0                 LastCId . . . . : 0
onlineBkupEnd . : 0                 softmax . . . . : 4194304


LOG RECORD - LSN <0:0:0:44162>
**********

HLG Header: lrecsize 212, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ALM Start Checkpoint (1025)
Eyecatcher  . . : ALRH              Version . . . . : 1
LogRecdLen  . . : 192               LogRecdOwnr . . : 1024   (ALM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

No data for Start Checkpoint Record
```

Figure 18. Example dmpmqlog output (Part 1 of 13)

```
LOG RECORD - LSN <0:0:0:44374>
**********

HLG Header: lrecsize 220, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Transaction Table (773)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 200                 LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 1
TranCount . . . : 0

LOG RECORD - LSN <0:0:0:44594>
**********

HLG Header: lrecsize 1836, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : Transaction Participants (1537)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 1816                LogRecdOwnr . . : 1536   (T)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Id. . . . . . . : TLPH
Version . . . . : 1                   Flags . . . . . : 3
Count . . . . . : 2

Participant Entry 0
RMName  . . . . : DB2 MQBankDB
RMId  . . . . . : 1
SwitchFile  . . : /Development/sbolam/build/devlib/tstxasw
XAOpenString  . :
XACloseString . :

Participant Entry 1
RMName  . . . . : DB2 MQBankDB
RMId  . . . . . : 2
SwitchFile  . . : /Development/sbolam/build/devlib/tstxasw
XAOpenString  . :
XACloseString . :
```

Figure 18. Example dmpmqlog output (Part 2 of 13)

```
LOG RECORD - LSN <0:0:0:46448>
**********

HLG Header: lrecsize 236, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ALM End Checkpoint (1026)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 216                 LogRecdOwnr . . : 1024   (ALM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

ChkPtLSN  . . . : <0:0:0:44162>
OldestLSN . . . : <0:0:0:0>
MediaLSN  . . . : <0:0:0:0>


LOG RECORD - LSN <0:0:0:52262>
**********

HLG Header: lrecsize 220, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Start Transaction (769)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 200                 LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: MQI    TranNum{High 0, Low 1}
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 1
SoftLogLimit  . : 10000
```

Figure 18. Example dmpmqlog output (Part 3 of 13)

```
LOG RECORD - LSN <0:0:0:52482>
**********

HLG Header: lrecsize 730, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : AQM Put Message (257)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 710                 LogRecdOwnr . . : 256    (AQM)
XTranid . . . . : TranType: MQI    TranNum{High 0, Low 1}
QueueName . . . : Queue1
Qid . . . . . . : {Hash 196836031, Counter: 0}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:52262>

Version . . . . : 3
SpcIndex  . . . : 1
PrevLink.Locn . : 36                  PrevLink.Length : 8
PrevDataLink  . : {High 0, Low 2048}
Data.Locn . . . : 2048                Data.Length . . : 486
Data  . . . . . :
00000:  41 51 52 48 00 00 00 04 FF FF FF FF FF FF FF FF    AQRH............
00016:  00 00 00 00 00 00 00 00 00 00 00 01 00 01 01 C0    ...............&#192;
00032:  00 00 00 00 00 00 00 01 00 00 00 22 00 00 00 00    ..........."....
00048:  00 00 00 00 41 4D 51 20 74 65 73 74 71 6D 20 20    ....AMQ testqm
00064:  20 20 20 20 33 80 2D D2 00 00 10 13 00 00 00 00        3€-&#30;........
00080:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00096:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00112:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01    ................
00128:  00 00 00 00 00 00 00 22 00 00 00 00 00 00 00 00    ......."........
00144:  00 00 00 00 00 00 00 C9 2C B5 C0 25 FF FF FF FF    .......&#26;,&#181;&#192;%....
00160:  4D 44 20 20 00 00 00 01 00 00 00 00 00 00 00 08    MD  ............
00176:  00 00 00 00 00 00 01 11 00 00 03 33 20 20 20 20    ...........3
00192:  20 20 20 20 00 00 00 00 00 00 00 01 20 20 20 20        ........
00208:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00224:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00240:  20 20 20 20 20 20 20 20 20 20 20 20 74 65 73 74                test
00256:  71 6D 20 20 20 20 20 20 20 20 20 20 20 20 20 20    qm
00272:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00288:  20 20 20 20 20 20 20 20 20 20 20 20 73 62 6F 6C                sbol
00304:  61 6D 20 20 20 20 20 20 04 37 34 38 30 00 00 00    am      .7480...
00320:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00336:  00 00 00 00 00 00 00 00 20 20 20 20 20 20 20 20    ........
00352:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00368:  20 20 20 20 20 20 20 20 00 00 00 06 75 74 7A 61            ....utza
00384:  70 69 20 20 20 20 20 20 20 20 20 20 20 20 20 20    pi
00400:  20 20 20 20 20 20 20 20 31 39 39 37 30 35 31 39            19970519
00416:  31 30 34 32 31 35 32 30 20 20 20 20 00 00 00 00    10421520    ....
00432:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00448:  50 65 72 73 69 73 74 65 6E 74 20 6D 65 73 73 61    Persistent messa
00464:  67 65 20 70 75 74 20 75 6E 64 65 72 20 73 79 6E    ge put under syn
00480:  63 70 6F 69 6E 74                                  cpoint
```

Figure 18. Example dmpmqlog output (Part 4 of 13)

```
LOG RECORD - LSN <0:0:0:53458>
**********

HLG Header: lrecsize 734, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : AQM Put Message (257)
Eyecatcher  . . : ALRH                 Version . . . . : 1
LogRecdLen  . . : 714                  LogRecdOwnr . . : 256    (AQM)
XTranid . . . . : TranType: NULL
QueueName . . . : Queue2
Qid . . . . . . : {Hash 184842943, Counter: 2}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 3
SpcIndex  . . . : 1
PrevLink.Locn . : 36                   PrevLink.Length : 8
PrevDataLink  . : {High 0, Low 2048}
Data.Locn . . . : 2048                 Data.Length . . : 490
Data  . . . . . :
00000:  41 51 52 48 00 00 00 04 FF FF FF FF FF FF FF FF    AQRH............
00016:  00 00 00 00 00 00 00 00 00 00 00 01 00 01 01 C0    ...............&#192;
00032:  00 00 00 00 00 00 00 01 00 00 00 26 00 00 00 00    ...........&;...
00048:  00 00 00 00 41 4D 51 20 74 65 73 74 71 6D 20 20    ....AMQ testqm
00064:  20 20 20 20 33 80 2D D2 00 00 10 13 00 00 00 00        3€-&#30;........
00080:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00096:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00112:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01    ................
00128:  00 00 00 00 00 00 00 26 00 00 00 00 00 00 00 00    .......&;.......
00144:  00 00 00 00 00 00 00 C9 2C B6 D8 DD FF FF FF FF    .......&#26;,.&#216;.....
00160:  4D 44 20 20 00 00 00 01 00 00 00 00 00 00 00 08    MD  ............
00176:  00 00 00 00 00 00 01 11 00 00 03 33 20 20 20 20    ...........3
00192:  20 20 20 20 00 00 00 00 00 00 00 01 20 20 20 20        ........
00208:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00224:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00240:  20 20 20 20 20 20 20 20 20 20 20 20 74 65 73 74                test
00256:  71 6D 20 20 20 20 20 20 20 20 20 20 20 20 20 20    qm
00272:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00288:  20 20 20 20 20 20 20 20 20 20 20 20 73 62 6F 6C                sbol
00304:  61 6D 20 20 20 20 20 20 04 37 34 38 30 00 00 00    am      .7480...
00320:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00336:  00 00 00 00 00 00 00 00 20 20 20 20 20 20 20 20    ........
00352:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00368:  20 20 20 20 20 20 20 20 00 00 00 06 75 74 7A 61            ....utza
00384:  70 69 20 20 20 20 20 20 20 20 20 20 20 20 20 20    pi
00400:  20 20 20 20 20 20 20 20 31 39 39 37 30 35 31 39            19970519
00416:  31 30 34 33 32 37 30 36 20 20 20 20 00 00 00 00    10432706    ....
00432:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00448:  50 65 72 73 69 73 74 65 6E 74 20 6D 65 73 73 61    Persistent messa
00464:  67 65 20 6E 6F 74 20 70 75 74 20 75 6E 64 65 72    ge not put under
00480:  20 73 79 6E 63 70 6F 69 6E 74                       syncpoint
```

Figure 18. Example dmpmqlog output (Part 5 of 13)

```
LOG RECORD - LSN <0:0:0:54192>
**********

HLG Header: lrecsize 216, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Commit Transaction (774)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 196                 LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: MQI    TranNum{High 0, Low 1}
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:52482>

Version . . . . : 1
LOG RECORD - LSN <0:0:0:54408>
**********

HLG Header: lrecsize 220, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Start Transaction (769)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 200                 LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: MQI    TranNum{High 0, Low 3}
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 1
SoftLogLimit  . : 10000

LOG RECORD - LSN <0:0:0:54628>
**********

HLG Header: lrecsize 240, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : AQM Get Message (259)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 220                 LogRecdOwnr . . : 256    (AQM)
XTranid . . . . : TranType: MQI    TranNum{High 0, Low 3}
QueueName . . . : Queue1
Qid . . . . . . : {Hash 196836031, Counter: 0}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:54408>

Version . . . . : 2
SpcIndex  . . . : 1                   QPriority . . . : 0
PrevLink.Locn . : 36                  PrevLink.Length : 8
PrevDataLink  . : {High 4294967295, Low 4294967295}
```

Figure 18. Example dmpmqlog output (Part 6 of 13)

```
LOG RECORD - LSN <0:0:0:54868>
**********

HLG Header: lrecsize 240, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : AQM Get Message (259)
Eyecatcher  . . : ALRH               Version . . . . : 1
LogRecdLen  . . : 220                LogRecdOwnr . . : 256    (AQM)
XTranid . . . . : TranType: NULL
QueueName . . . : Queue2
Qid . . . . . . : {Hash 184842943, Counter: 2}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 2
SpcIndex  . . . : 1                  QPriority . . . : 0
PrevLink.Locn . : 36                 PrevLink.Length : 8
PrevDataLink  . : {High 4294967295, Low 4294967295}
LOG RECORD - LSN <0:0:0:55108>
**********

HLG Header: lrecsize 216, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Commit Transaction (774)
Eyecatcher  . . : ALRH               Version . . . . : 1
LogRecdLen  . . : 196                LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: MQI    TranNum{High 0, Low 3}
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:54628>

Version . . . . : 1

LOG RECORD - LSN <0:0:0:55324>
**********

HLG Header: lrecsize 220, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Start Transaction (769)
Eyecatcher  . . : ALRH               Version . . . . : 1
LogRecdLen  . . : 200                LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: XA
   XID: formatID 5067085, gtrid_length 14, bqual_length 4
        gtrid [3270BDB4000010237465737471 6D]
        bqual [00000001]
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 1
SoftLogLimit  . : 10000
```

Figure 18. Example dmpmqlog output (Part 7 of 13)

```
LOG RECORD - LSN <0:0:0:55544>
**********

HLG Header: lrecsize 738, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : AQM Put Message (257)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 718                 LogRecdOwnr . . : 256    (AQM)
XTranid . . . . : TranType: XA
   XID: formatID 5067085, gtrid_length 14, bqual_length 4
        gtrid [3270BDB40000102374657374716D]
        bqual [00000001]
QueueName . . . : Queue2
Qid . . . . . . : {Hash 184842943, Counter: 2}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:55324>

Version . . . . : 3
SpcIndex  . . . : 1
PrevLink.Locn . : 36                  PrevLink.Length : 8
PrevDataLink  . : {High 0, Low 2048}
Data.Locn . . . : 2048                Data.Length . . : 494
Data  . . . . . :
00000:  41 51 52 48 00 00 00 04 FF FF FF FF FF FF FF FF    AQRH............
00016:  00 00 00 00 00 00 00 00 00 00 00 01 00 01 01 C0    ...............&#192;
00032:  00 00 00 00 00 00 00 01 00 00 00 2A 00 00 00 00    ...........*....
00048:  00 00 00 01 41 4D 51 20 74 65 73 74 71 6D 20 20    ....AMQ testqm
00064:  20 20 20 20 33 80 2D D2 00 00 10 13 00 00 00 00        3€-&#30;........
00080:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00096:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00112:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01    ................
00128:  00 00 00 00 00 00 00 2A 00 00 00 00 00 00 00 00    .......*........
00144:  00 00 00 00 00 00 00 C9 2C B8 3E E8 FF FF FF FF    .......&#26;,&#184;>.....
00160:  4D 44 20 20 00 00 00 01 00 00 00 00 00 00 00 08    MD  ............
00176:  00 00 00 00 00 00 01 11 00 00 03 33 20 20 20 20    ...........3
00192:  20 20 20 20 00 00 00 00 00 00 00 01 20 20 20 20        ........
00208:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00224:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00240:  20 20 20 20 20 20 20 20 20 20 20 20 74 65 73 74                test
00256:  71 6D 20 20 20 20 20 20 20 20 20 20 20 20 20 20    qm
00272:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00288:  20 20 20 20 20 20 20 20 20 20 20 20 73 62 6F 6C                sbol
00304:  61 6D 20 20 20 20 20 20 04 37 34 38 30 00 00 00    am      .7480...
00320:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00336:  00 00 00 00 00 00 00 00 20 20 20 20 20 20 20 20    ........
00352:  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00368:  20 20 20 20 20 20 20 20 00 00 00 06 75 74 7A 61            ....utza
00384:  70 69 20 20 20 20 20 20 20 20 20 20 20 20 20 20    pi
00400:  20 20 20 20 20 20 20 20 31 39 39 37 30 35 31 39            19970519
00416:  31 30 34 34 35 38 37 32 20 20 20 20 00 00 00 00    10445872    ....
00432:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00448:  41 6E 6F 74 68 65 72 20 70 65 72 73 69 73 74 65    Another persiste
00464:  6E 74 20 6D 65 73 73 61 67 65 20 70 75 74 20 75    nt message put u
00480:  6E 64 65 72 20 73 79 6E 63 70 6F 69 6E 74          nder syncpoint
```

Figure 18. Example dmpmqlog output (Part 8 of 13)

```
LOG RECORD - LSN <0:0:0:56282>
**********

HLG Header: lrecsize 216, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Prepare Transaction (770)
Eyecatcher  . . : ALRH                 Version . . . . : 1
LogRecdLen  . . : 196                  LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: XA
   XID: formatID 5067085, gtrid_length 14, bqual_length 4
        gtrid [3270BDB40000102374657374716D]
        bqual [00000001]
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:55544>

Version . . . . : 1


LOG RECORD - LSN <0:0:0:56498>
**********

HLG Header: lrecsize 708, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : Transaction Prepared (1538)
Eyecatcher  . . : ALRH                 Version . . . . : 1
LogRecdLen  . . : 688                  LogRecdOwnr . . : 1536   (T)
XTranid . . . . : TranType: XA
   XID: formatID 5067085, gtrid_length 14, bqual_length 4
        gtrid [3270BDB40000102374657374716D]
        bqual [00000001]
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Id. . . . . . . : TLPR
Version . . . . : 1                    Flags . . . . . : 1
Count . . . . . : 3

Participant Entry 0
RMId  . . . . . : 0                    State . . . . . : 2

Participant Entry 1
RMId  . . . . . : 1                    State . . . . . : 2

Participant Entry 2
RMId  . . . . . : 2                    State . . . . . : 2
```

Figure 18. Example dmpmqlog output (Part 9 of 13)

```
LOG RECORD - LSN <0:0:0:57206>
**********

HLG Header: lrecsize 216, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Commit Transaction (774)
Eyecatcher  . . : ALRH               Version . . . . : 1
LogRecdLen  . . : 196                LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: XA
   XID: formatID 5067085, gtrid_length 14, bqual_length 4
        gtrid [3270BDB4000010237465737471 6D]
        bqual [00000001]
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:56282>

Version . . . . : 1
LOG RECORD - LSN <0:0:0:57440>
**********

HLG Header: lrecsize 224, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : Transaction Forget (1539)
Eyecatcher  . . : ALRH               Version . . . . : 1
LogRecdLen  . . : 204                LogRecdOwnr . . : 1536   (T)
XTranid . . . . : TranType: XA
   XID: formatID 5067085, gtrid_length 14, bqual_length 4
        gtrid [3270BDB40000102374657374716D]
        bqual [00000001]
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Id. . . . . . . : TLFG
Version . . . . : 1                  Flags . . . . . : 0
```

Figure 18. Example dmpmqlog output (Part 10 of 13)

```
LOG RECORD - LSN <0:0:0:58120>
**********

HLG Header: lrecsize 212, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ALM Start Checkpoint (1025)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 192                 LogRecdOwnr . . : 1024   (ALM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

No data for Start Checkpoint Record

LOG RECORD - LSN <0:0:0:58332>
**********

HLG Header: lrecsize 220, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ATM Transaction Table (773)
Eyecatcher  . . : ALRH                Version . . . . : 1
LogRecdLen  . . : 200                 LogRecdOwnr . . : 768    (ATM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 1
TranCount . . . : 0
```

Figure 18. Example dmpmqlog output (Part 11 of 13)

```
LOG RECORD - LSN <0:0:0:58552>
**********

HLG Header: lrecsize 1836, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : Transaction Participants (1537)
Eyecatcher  . . : ALRH                 Version . . . . : 1
LogRecdLen  . . : 1816                 LogRecdOwnr . . : 1536   (T)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Id. . . . . . . : TLPH
Version . . . . : 1                    Flags . . . . . : 3
Count . . . . . : 2

Participant Entry 0
RMName  . . . . : DB2 MQBankDB
RMId  . . . . . : 1
SwitchFile  . . : /Development/sbolam/build/devlib/tstxasw
XAOpenString  . :
XACloseString . :

Participant Entry 1
RMName  . . . . : DB2 MQFeeDB
RMId  . . . . . : 2
SwitchFile  . . : /Development/sbolam/build/devlib/tstxasw
XAOpenString  . :
XACloseString . :

LOG RECORD - LSN <0:0:0:60388>
**********

HLG Header: lrecsize 236, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ALM End Checkpoint (1026)
Eyecatcher  . . : ALRH                 Version . . . . : 1
LogRecdLen  . . : 216                  LogRecdOwnr . . : 1024   (ALM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

ChkPtLSN  . . . : <0:0:0:58120>
OldestLSN . . . : <0:0:0:0>
MediaLSN  . . . : <0:0:0:0>
```

Figure 18. Example dmpmqlog output (Part 12 of 13)

```
LOG RECORD - LSN <0:0:0:60624>
**********

HLG Header: lrecsize 240, version 1, rmid 0, eyecatcher HLRH

LogRecdType . . : ALM Stop Queue Manager (1028)
Eyecatcher  . . : ALRH               Version . . . . : 1
LogRecdLen  . . : 220                LogRecdOwnr . . : 1024   (ALM)
XTranid . . . . : TranType: NULL
QueueName . . . : NULL
Qid . . . . . . : {NULL_QID}
ThisLSN . . . . : <0:0:0:0>
PrevLSN . . . . : <0:0:0:0>

Version . . . . : 1
StopDate  . . . : 19970519           StopTime  . . . : 10490868
SessionNumber . : 0                  ForceFlag . . . : Quiesce

AMQ7702: DMPMQLOG command has finished successfully.
```

Figure 18. Example dmpmqlog output (Part 13 of 13)

Notes for Figure 18 on page 143:

1. The *headlsn* in the *Log File Header* has a value of <0:0:0:58120>. This is where the dump would have started had we not requested a different starting LSN.
2. The *nextlsn* is <0:0:0:60864> which will be the LSN of the first log record that the queue manager will write when it is next restarted.
3. The *HeadExtentID* is 1, indicating that the head of the log currently resides in log file S0000001.LOG.
4. The first log record formatted is a *Start Checkpoint* log record. The checkpoint spans a number of log records until the *End CheckPoint* record at <0:0:0:46448>.
5. One of the records logged during checkpoint is the *Transaction Participants* log record at <0:0:0:44594>. This details the resource managers that participate in global transactions coordinated by the queue manager.
6. The *Start Transaction* log record at <0:0:0:52262> denotes the start of a transaction. The *XTranid* shows a *TranType* of MQI, which indicates that it is a local transaction including MQ updates only.
7. The next log record is a *Put Message* log record that records the persistent MQPUT under the syncpoint that started the transaction. The MQPUT was made to the queue *Queue1* and the message data is logged as `Persistent message put under syncpoint`. This message has been allocated a *SpcIndex* of 1, which will be matched to the later MQGET of this message.
8. The next log record at LSN <0:0:0:53458> is also a *Put Message* record. This persistent message was put to a different queue, *Queue2*, but was not made under syncpoint since the *XTranid* is *NULL*. It too has a *SpcIndex* of 1, which is a unique identifier for this particular queue.
9. The next log record at LSN <0:0:0:54192> commits the message that was put under syncpoint.
10. In log records <0:0:0:54408> and <0:0:0:54628> a new transaction is started by an MQGET under syncpoint for queue *Queue1*. The *SpcIndex* in the *Get Message* log record is 1 indicating that this was the same message that was put to *Queue1* in <0:0:0:52262>.
11. The next log record gets the message that was put to *Queue2* by the other *Put Message* log record.

12. The MQGET under syncpoint has been committed as indicated by the *Commit Transaction* log record at <0:0:0:55108>.
13. Finally an MQBEGIN is used to start a global transaction in the *Start Transaction* log record at <0:0:0:55324>. The *XTranid* in this log record has a *TranType* of XA.
14. The following *Put Message* records a persistent message put to *Queue2*. This shares the same *XTranid* as the previous log record.
15. If a *Transaction Prepared* log record is written for this *Xtranid* then the transaction as a whole must be committed. The absence of such a log record can be taken as an indication that the transaction was rolled back. In this case a *Transaction Prepared* log record is found at <0:0:0:56498>. This records the queue manager itself as a participant with an *RMId* of zero. There are two further participants, their *RMIds* of 1 and 2 can be matched with the previous *Transaction Participants* log record.
16. During the commit phase the XA Transaction Manager component of the queue manager does not log individual responses from the participants. The log indicates only whether the queue manager updates were committed or not. The *Commit Transaction* log record at <0:0:0:57206> indicates that the message was indeed committed to *Queue2*.
17. The *Transaction Forget* log record at <0:0:0:57440> indicates that the commit decision was also delivered to the other two resource managers. Any failure of these resource managers to commit their updates will have been diagnosed in the queue manager’s error logs.

Chapter 12. Using the name service

The name service is an installable service that enables an application connected to one queue manager to open what it thinks are local queues. These queues are actually queues defined on another queue manager – on another machine – with the SCOPE attribute set to CELL.

The application can perform all the operations permitted for remote queues on queues opened in this way. The supplied implementation uses DCE (Distributed Computing Environment), although you are free to write your own component that does not use DCE.

To use the supplied name service component, you must define the name service and its installed component to the queue manager. You do this by inserting the appropriate stanza in the queue manager configuration file (qm.ini) file. See the *MQSeries Programmable System Management* book for details. You will also need to do some DCE configuration.

Using DCE to share queues on different queue managers

If your queue managers are located on nodes within a Distributed Computing Environment (DCE) cell, you can configure them to share queues. Applications can then connect to one queue manager and open a queue on *another* queue manager on another node. To the application, this is transparent; it is not aware that the queue actually resides on another queue manager. (Normally, the queue manager rejects open requests from a local application if the queue does not exist on that queue manager.)

Configuration tasks for shared queues

This section describes how you set up shared queues on queue managers that reside on nodes that are within the DCE cell.

For each queue manager:

1. Configure the name service by adding the required name service stanza to the queue manager configuration file. The contents of this stanza are described in *MQSeries Programmable System Management* book. To invoke the name service, you have to restart the queue manager.
2. Use the **endmqm** command to stop the queue manager if it is running.
3. Use the **strmqm** command to restart the queue manager.
4. Set up channels for messaging between queue managers; see "Preparing channels and transmission queues for remote administration" on page 62.

Sharing queues

For any queue that you want to be shared, specify the SCOPE attribute as CELL. For example, use these MQSC commands:

```
DEFINE QLOCAL(GREY.PUBLIC.QUEUE) SCOPE(CELL)
```

or

```
ALTER QLOCAL(PINK.LOCAL.QUEUE) SCOPE(CELL)
```

The queue created or altered must belong to a queue manager on a node within the DCE cell.

DCE configuration

To use the supplied name service component, you must have the OSF Distributed Computing Environment (DCE) installed. This service enables applications that connect to one queue manager to open queues that belong to another queue manager in the same DCE cell.

An example DCL shell script, that allows the supplied name service to run, is supplied in the `mqs_examples:dcesetu.com`.

Chapter 13. Configuring MQSeries

This chapter explains how to change the behavior of an individual queue manager, or of a node, to suit your installation's needs.

You change MQSeries configuration information by modifying the values specified on a set of configuration attributes (or parameters) which govern MQSeries.

How you change this configuration information, and where MQSeries stores your changes, is platform-specific. Users of MQSeries for Compaq OpenVMS change the configuration information by editing the **MQSeries configuration files**.

This chapter:
- Describes the attributes you can use to modify MQSeries configuration information in "Attributes for changing MQSeries configuration information" on page 161.
- Describes the attributes you can use to modify queue manager configuration information in "Changing queue manager configuration information" on page 166.
- Provides examples of `mqs.ini` and `qm.ini` files for MQSeries for Compaq OpenVMS in "Example mqs.ini and qm.ini files" on page 173.

MQSeries configuration files

Users of MQSeries for Compaq OpenVMS modify MQSeries configuration attributes within:
- An MQSeries configuration file (`mqs.ini`) to make changes to MQSeries on the node as a whole. There is one `mqs.ini` file per node.
- A queue manager configuration file (`qm.ini`) to make changes to specific queue managers. There is one `qm.ini` file for each queue manager on the node.

A configuration file (which may also be referred to as a ***stanza*** file or ***.ini*** file) contains one or more stanzas, which are simply groups of lines in the file that together have a common function or define part of a system, for example, log functions, channel functions, and installable services.

Any changes you make to a configuration file will not take effect until the next time the queue manager is started.

Editing configuration files

Before attempting to edit a configuration file, back it up so that you have a copy you can revert to if the need arises!

You can edit configuration files either:
- Automatically, using commands that change the configuration of queue managers on the node
- Manually, using a standard text editor

You can edit the default values in the MQSeries configuration files after installation.

If you set an incorrect value on a configuration file attribute, the value is ignored and an operator message is issued to indicate the problem. (The effect is the same as missing out the attribute entirely.)

When you create a new queue manager, you should:
- Back up the MQSeries configuration file
- Back up the new queue manager configuration file

When do you need to edit a configuration file?

You may need to edit a configuration file if, for example:
- You lose a configuration file; recover from backup if possible.
- You need to move one or more queue managers to a new directory.
- You need to change your default queue manager; this could happen if you accidentally delete the existing queue manager.
- You are advised to do so by your IBM Support Center.

Configuration file priorities

The attribute values of a configuration file are set according to the following priorities:
- Parameters entered on the command line take precedence over values defined in the configuration files
- Values defined in the `qm.ini` files take precedence over values defined in the `mqs.ini` file.

Implementing changes to configuration files

If you edit a configuration file, the changes are not implemented immediately by the queue manager. Changes made to the MQSeries configuration file are only implemented when MQSeries is started. Changes made to a queue manager configuration file are implemented when the queue manager is started. If the queue manager is running when you make the changes, you must stop and then restart the queue manager for any changes to be recognized by the system.

The MQSeries configuration file, mqs.ini

The MQSeries configuration file, `mqs.ini`, contains information relevant to all the queue managers on the node. It is created automatically during installation. In particular, the `mqs.ini` file is used to locate the data associated with each queue manager.

The `mqs.ini` file is stored in the data directory by default, **MQS_ROOT:[MQM]**.

The `mqs.ini` file contains:
- The names of the queue managers
- The name of the default queue manager
- The location of the files associated with each of them.

For more information on `mqs.ini` contents, see “Attributes for changing MQSeries configuration information” on page 161.

Queue manager configuration files, qm.ini

A queue manager configuration file, `qm.ini`, contains information relevant to a specific queue manager. There is one queue manager configuration file for each queue manager. The `qm.ini` file is automatically created when the queue manager with which it is associated is created.

A `qm.ini` file is held in the root of the directory tree occupied by the queue manager.

For example, in MQSeries for Compaq OpenVMS, the path and the name for a configuration file for a queue manager called QMNAME is:

```
MQS_ROOT:[MQM.QMGRS.QMNAME]QM.INI
```

Note: The queue manager name can be up to 48 characters in length. However, this does not guarantee that the name is valid or unique. Therefore, a directory name is generated based on the queue manager name. This process is known as **name transformation**. For a description, see “Understanding MQSeries file names” on page 19.

For more information about `qm.ini`, see “Changing queue manager configuration information” on page 166.

Attributes for changing MQSeries configuration information

The following groups of attributes appear in `mqs.ini`:
- The AllQueueManagers stanza
- “The ClientExitPath stanza” on page 162
- “The DefaultQueueManager stanza” on page 162
- “The ExitProperties stanza” on page 163
- “The LogDefaults stanza” on page 163
- “The QueueManager stanza” on page 165

A sample `mqs.ini` is shown in “Example mqs.ini and qm.ini files” on page 173.

The AllQueueManagers stanza

The `AllQueueManagers` stanza can specify:
- The path to the qmgrs directory where the files associated with a queue manager are stored
- The method for converting EBCDIC-format data to ASCII format

DefaultPrefix=*directory_name*

This attribute specifies the path to the qmgrs directory, below which the queue manager data is kept.

If you change the default prefix for the queue manager, you must replicate the directory structure that was created at installation time (see “Appendix C. Directory structure” on page 299).

In particular, the qmgrs structure must be created. You must stop MQSeries before changing the default prefix, and restart MQSeries only after the structures have been moved to the new location and the default prefix has been changed.

As an alternative to changing the default prefix, you can use the logical MQSPREFIX to override the `DefaultPrefix` for the **crtmqm** command.

ConvEBCDICNewline=NL_TO_LF|TABLE|ISO

EBCDIC code pages contain a new line (NL) character that is not supported by ASCII code pages; although some ISO variants of ASCII do contain an equivalent.

Use the ConvEBCDICNewline attribute to specify the method MQSeries is to use when converting the EBCDIC NL character into ASCII format.

NL_TO_LF
Specify NL_TO_LF if you want the EBCDIC NL character (X'15') converted to the ASCII line feed character, LF (X'0A'), for all EBCDIC to ASCII conversions.

NL_TO_LF is the default.

TABLE
Specify TABLE if you want the EBCDIC NL character converted according to the conversion tables used on your platform for all EBCDIC to ASCII conversions.

Note that the effect of this type of conversion may vary from platform to platform and from language to language; while on the same platform, the behavior may vary if you use different CCSIDs.

ISO
Specify ISO if you want:

- ISO CCSIDs to be converted using the TABLE method
- All other CCSIDs to be converted using the NL_TO_LF method.

Possible ISO CCSIDs are shown in Table 7.

Table 7. List of possible ISO CCSIDs

CCSID	Code Set
819	ISO8859-1
912	ISO8859-2
915	ISO8859-5
1089	ISO8859-6
813	ISO8859-7
916	ISO8859-8
920	ISO8859-9
1051	roman8

If the ASCII CCSID is not an ISO subset, ConvEBCDICNewline defaults to NL_TO_LF.

For more information about data conversion, see the *MQSeries Application Programming Guide* or “Data conversion” on page 71.

The ClientExitPath stanza

The `ClientExitPath` stanza specifies the default path for location of the channel exit on the client.

ExitsDefaultPath=*defaultprefix*
The ExitsDefaultPath attribute specifies the default prefix for the platform.

The DefaultQueueManager stanza

The `DefaultQueueManager` stanza specifies the default queue manager for the node.

Name=*default_queue_manager*
The default queue manager processes any commands for which a queue manager name is not explicitly specified. The `DefaultQueueManager` attribute is automatically updated if you create a new default queue manager. If you

inadvertently create a new default queue manager and then want to revert to the original, you must alter the `DefaultQueueManager` attribute manually.

The ExitProperties stanza

The `ExitProperties` stanza specifies configuration options used by queue manager exit programs.

CLWLMode=SAFE|FAST

The cluster workload exit, CLWL, allows you to specify which cluster queue in the cluster is to be opened in response to an MQAPI call (MQOPEN or MQPUT and so on). The CLWL exit runs either in FAST mode or SAFE mode depending on the value you specify on the CLWLMode attribute. If the CLWLMode attribute is not specified, the cluster workload exit runs in SAFE mode.

SAFE

The SAFE option specifies that the CLWL exit is to run in a separate process to the queue manager. This is the default.

If a problem arises with the user-written CLWL exit when running in SAFE mode, the following happens:

- The CLWL server process (amqzlwa0) fails
- The queue manager restarts the CLWL server process
- The error is reported to you in the error log. If an MQAPI call is in progress, you receive notification in the form of a bad return code.

The integrity of the queue manager is preserved.

Note: There is an overhead associated with running the CLWL exit in a separate process, which can affect performance.

FAST

Specify FAST if you want the cluster exit to run inline in the queue manager process.

Specifying this option improves performance by avoiding the overheads associated with running in SAFE mode, but does so at the expense of queue manager integrity. Therefore, you should only run the CLWL exit in FAST mode if you are convinced that there are **no** problems with your CLWL exit, and you are particularly concerned about performance overheads.

If a problem arises when the CLWL exit is running in FAST mode, the queue manager will fail and you run the risk of the integrity of the queue manager being compromised.

The LogDefaults stanza

The `LogDefaults` stanza specifies the default log attributes for the node. The log attributes are used as default values when you create a queue manager, but can be overridden if you specify the log attributes on the **crtmqm** command. See “crtmqm (Create queue manager)” on page 231 for details of this command.

Once a queue manager has been created, the log attributes for that queue manager are read from its log stanza in the `qm.ini` file.

Changing MQSeries configuration file

The *DefaultPrefix* attribute (in the `AllQueueManagers` stanza) and the *LogPath* attribute in the `LogDefaults` stanza allow for the queue manager and its log to be on different physical drives. This is the recommended method, although, by default, they are on the same drive.

For information about calculating log sizes, see "Calculating size of log" on page 130.

Note: The limits given in the following parameter list are limits set by MQSeries. Operating system limits may reduce the maximum possible log size.

LogPrimaryFiles=3|*2-62*
Primary log files are the log files allocated during creation for future use.

The minimum number of primary log files you can have is 2 and the maximum is 62. The default is 3.

The total number of primary and secondary log files must not exceed 63, and must not be less than 3.

This value is overwritten by the -lp parameter of the **crtmqm** command when the queue manager is created.

LogSecondaryFiles=2|*1-61*
Secondary log files are the log files allocated when the primary files are exhausted.

The minimum number of secondary log files is 1 and the maximum is 61. The default number is 2.

The total number of primary and secondary log files must not exceed 63, and must not be less than 3.

This value is overwritten by the -ls parameter of the **crtmqm** command when the queue manager is created.

LogFilePages=*number*
The log data is held in a series of files called log files. The log file size is specified in units of 4 KB pages.

For MQSeries for Compaq OpenVMS, the default number of log file pages is 1024, giving a log file size of 4 MB. The minimum number of log file pages is 64 and the maximum is 16 384.

This value is overwritten by the -lf parameter of the **crtmqm** command when the queue manager is created.

LogType=CIRCULAR|LINEAR
The `LogType` attribute is used to define the type to be used. The default is CIRCULAR.

CIRCULAR
Set this value if you want to start restart recovery using the log to roll back transactions that were in progress when the system stopped.

See "Circular logging" on page 126 for a fuller explanation of circular logging.

LINEAR
Set this value if you want both restart recovery and media or forward recovery (creating lost or damaged data by replaying the contents of the log).

See "Linear logging" on page 126 for a fuller explanation of linear logging.

If you want to change the default logtype, you can edit the `LogType` attribute in the `mqs.ini` file. Alternatively, you can override the default by specifying linear logging using the -ll parameter on the **crtmqm** command. You cannot change the logging method after a queue manager has been created.

LogBufferPages=17|*4-32*
The amount of memory allocated to buffer records for writing is configurable. The size of the buffers is specified in units of 4 KB pages.

The minimum number of buffer pages is 4 and the maximum is 32. Larger buffers lead to higher throughput, especially for larger messages.

The default number of buffer pages is 17, equating to 68 KB.

The value is examined when the queue manager is created or started, and may be increased or decreased at either of these times. However, a change in the value is not effective until the queue manager is restarted.

LogDefaultPath=*directory_name*
You can specify the directory in which the log files for a queue manager reside. The directory should exist on a local device to which the queue manager can write and, preferably, should be on a different drive from the message queues. Specifying a different drive gives added protection in case of system failure.

The default for MQSeries for Compaq OpenVMS is `MQS_ROOT:[MQM.LOG]`.

Alternatively, you can specify the name of a directory on the **crtmqm** command using the -ld flag. When a queue manager is created, a directory is also created under the queue manager directory, and this is used to hold the log files. The name of this directory is based on the queue manager name. This ensures that the Log File Path is unique, and also that it conforms to any limitations on directory name lengths.

If you do not specify -ld on the **crtmqm** command, the value of the `LogDefaultPath` attribute in the `mqs.ini` file is used by default and this is `MQS_ROOT:[MQM.LOG]`.

The queue manager name is appended to the log file directory name to ensure that multiple queue managers use different log directories.

When the queue manager has been created, a `LogPath` value is created in the Log stanza in the `qm.ini` file giving the complete directory name for the queue manager's log files. This value is used to locate the log files when the queue manager is started or deleted.

The QueueManager stanza

There is one `QueueManager` stanza for every queue manager. These attributes specify the queue manager name, and the name of the directory containing the files associated with that queue manager. The name of the directory is based on the queue manager name, but is transformed if the queue manager name is not a valid file name.

See "Understanding MQSeries file names" on page 19 for more information about name transformation.

Name=*queue_manager_name*
This attribute specifies the name of the queue manager.

Prefix=*prefix*
This attribute specifies where the queue manager files are stored. By default,

this is the same as the value specified on the DefaultPrefix attribute of the `AllQueueManager` stanza in the `mqs.ini` file.

Directory=*name*
: This attribute specifies the name of the subdirectory where the queue manager files are stored. This will normally be under `MQS_ROOT:[MQM.QMGRS]` unless an alternative prefix value has been specified. This name is based on the queue manager name but can be transformed if there is a duplicate name, or if the queue manager name is not a valid file name.

Changing queue manager configuration information

The following groups of attributes can appear in a `qm.ini` file particular to a given queue manager, or used to override values set in `mqs.ini`.

- "The Service stanza"
- "The ServiceComponent stanza" on page 167
- "The Log stanza" on page 167
- "The XAResourceManager stanza" on page 169
- "The Channels stanza" on page 170
- "The LU62 and TCP stanzas" on page 172
- "The ExitPath stanza" on page 173

The Service stanza

The `Service` stanza specifies the name of an installable service, and the number of entry points to that service. There must be one `Service` stanza for every service used.

For each component within a service, there must be a `ServiceComponent` stanza, which identifies the name and path of the module containing the code for that component. See The ServiceComponent stanza for more information.

Name=AuthorizationService|NameService
: Specifies the name of the required service.

AuthorizationService
: For MQSeries, the Authorization Service component is known as the Object Authority Manager, or OAM.

In MQSeries for Compaq OpenVMS, the `AuthorizationService` stanza and its associated `ServiceComponent` stanza are added automatically when the queue manager is created, but can be overridden through the use of `mqsnoaut`, by setting the `mqsnoaut` logical before creating the queue manager. (See "Disabling the object authority manager" on page 76 for more information.) Any other `ServiceComponent` stanzas must be added manually.

NameService
: The `NameService` stanza must be added to the `qm.ini` file manually to enable the supplied name service.

EntryPoints=*number-of-entries*
: Specifies the number of entry points defined for the service. This includes the initialization and termination entry points.

For more information about installable services and components, see the *MQSeries Programmable System Management* book.

For more information about security services in general, see “Chapter 7. Protecting MQSeries objects” on page 73.

The ServiceComponent stanza

The `ServiceComponent` stanza identifies the name and path of the module containing the code for that component.

There can be more than one ServiceComponent stanza for each service, but each ServiceComponent stanza must match the corresponding Service stanza.

In MQSeries for Compaq OpenVMS, the authorization service stanza is present by default, and the associated component, the OAM, is active.

Service=*service_name*
: Specifies the name of the required service. This name must match the value specified on the `Name` attribute of the `Service` stanza.

Name=*component_name*
: Specifies the descriptive name of the service component. This name must be unique, and must contain only those characters that are valid for the names of MQSeries objects (for example, queue names). This name occurs in operator messages generated by the service. It is recommended, therefore, that this name begins with a company trademark or similar distinguishing string.

Module=*module_name*
: Specifies the name of the module to contain the code for this component.

  Note: Specify a full path name.

ComponentDataSize=*size*
: Specifies the size, in bytes, of the component data area passed to the component on each call. Specify zero if no component data is required.

For more information about installable services and components, see the *MQSeries Programmable System Management* book.

The Log stanza

The `Log` stanza specifies the log attributes for a particular queue manager. By default, these are inherited from the settings specified in the `LogDefaults` stanza in the `mqs.ini` file when the queue manager is created, unless overridden by specific parameters in the **crtmqm** command. For more information, see both “The LogDefaults stanza” on page 163 and “crtmqm (Create queue manager)” on page 231.

Only change attributes of this stanza if this particular queue manager needs to be configured differently from your other ones.

The values specified on the attributes in the `qm.ini` file are read when the queue manager is started. The file is created when the queue manager is created.

For information about calculating log sizes, see “Calculating size of log” on page 130.

Note: The limits given in the following parameter list are limits set by MQSeries. Operating system limits may reduce the maximum possible log size.

LogPrimaryFiles=3|*2-62*
Primary log files are the log files allocated during creation for future use.

The minimum number of primary log files you can have is 2 and the maximum is 62. The default is 3.

The total number of primary and secondary log files must not exceed 63, and must not be less than 3.

The value is examined when the queue manager is created or started. You can change it after the queue manager has been created. However, a change in the value is not effective until the queue manager is restarted, and the effect may not be immediate.

LogSecondaryFiles=2|*1-61*
Secondary log files are the log files allocated when the primary files are exhausted.

The minimum number of secondary log files is 1 and the maximum is 61. The default number is 2.

The total number of primary and secondary log files must not exceed 63, and must not be less than 3.

The value is examined when the queue manager is started. You can change this value, but changes do not become effective until the queue manager is restarted, and even then the effect may not be immediate.

LogFilePages=*number*
The log data is held in a series of files called log files. The log file size is specified in units of 4 KB pages.

In MQSeries for Compaq OpenVMS, the default number of log file pages is 1024, giving a log file size of 4 MB. The minimum number of log file pages is 64 and the maximum is 16 384.

Note: The size of the log files specified during queue manager creation cannot be changed for an existing queue manager.

LogType=CIRCULAR|LINEAR
The `LogType` attribute defines the type of logging to be used by the queue manager. However, you cannot change the type of logging to be used once the queue manager has been created. Refer to the description of the LogType attribute in “The LogDefaults stanza” on page 163 for information about creating a queue manager with the type of logging you require.

CIRCULAR
Set this value if you want to start restart recovery using the log to roll back transactions that were in progress when the system stopped.

See “Circular logging” on page 126 for a fuller explanation of circular logging.

LINEAR
Set this value if you want both restart recovery and media or forward recovery (creating lost or damaged data by replaying the contents of the log).

See “Linear logging” on page 126 for a fuller explanation of linear logging.

LogBufferPages=17|*4-32*
The amount of memory allocated to buffer records for writing is configurable. The size of the buffers is specified in units of 4 KB pages.

The minimum number of buffer pages is 4 and the maximum is 32. Larger buffers lead to higher throughput, especially for larger messages.

The default number of buffer pages is 17, equating to 68 KB.

The value is examined when the queue manager is started, and may be increased or decreased at either of these times. However, a change in the value is not effective until the queue manager is restarted.

LogPath=*directory_name*

You can specify the directory in which the log files for a queue manager reside. The directory should exist on a local device to which the queue manager can write and, preferably, should be on a different drive from the message queues. Specifying a different drive gives added protection in case of system failure.

The default is `MQS_ROOT:[MQM.LOG]`.

You can specify the name of a directory on the **crtmqm** command using the -ld flag. When a queue manager is created, a directory is also created under the queue manager directory, and this is used to hold the log files. The name of this directory is based on the queue manager name. This ensures that the log file path is unique, and also that it conforms to any limitations on directory name lengths.

If you do not specify -ld on the **crtmqm** command, the value of the `LogDefaultPath` attribute in the `mqs.ini` file is used.

Note: In MQSeries for Compaq OpenVMS, user ID mqm and group mqm must have full authorities to the log files. If you change the locations of these files, you must give these authorities yourself. This is not required if the log files are in the default locations supplied with the product.

The XAResourceManager stanza

The `XAResourceManager` stanza specifies the resource managers to be involved in global units of work coordinated by the queue manager.

One `XAResourceManager` stanza is required in `qm.ini` for each instance of a resource manager participating in global units of work; no default values are supplied via `mqs.ini`.

See "Database coordination" on page 112 for more information about adding `XAResourceManager` attributes to `qm.ini`.

Name=*name* **(mandatory)**

This attribute identifies the resource manager instance.

The `Name` value can be up to 31 characters in length and must be unique within `qm.ini`. You can use the name of the resource manager as defined in its XA-switch structure. However, if you are using more than one instance of the same resource manager, you must construct a unique name for each instance. You could ensure uniqueness by including the name of the database in the `Name` string, for example.

MQSeries uses the `Name` value in messages and in output from the **dspmqtrn** command.

You are recommended not to change the name of a resource manager instance, or to delete its entry from `qm.ini` once the associated queue manager has started and the resource-manager name is in effect.

SwitchFile=*name* **(mandatory)**
This attribute specifies the fully-qualified name of the load file containing the resource manager's XA switch structure.

XAOpenString=*string* **(optional)**
This attribute specifies the string of data to be passed to the resource manager's xa_open entry point. The contents of the string depend on the resource manager itself. For example, the string could identify the database that this instance of the resource manager is to access. For more information about defining this attribute, see "Adding XAResourceManager configuration information for Oracle" on page 117 and consult your resource manager documentation for the appropriate string.

XACloseString=*string* **(optional)**
This attribute specifies the string of data to be passed to the resource manager's xa_close entry point. The contents of the string depend on the resource manager itself. For more information about defining this attribute, see"Adding XAResourceManager configuration information for Oracle" on page 117 and consult your database documentation for the appropriate string.

ThreadOfControl=THREAD|PROCESS
The value set on the ThreadOfControl attribute is used by the queue manager for serialization purposes when it needs to call the resource manager from one of its own multithreaded processes.

THREAD
Means that the resource manager is fully "thread aware". In a multithreaded MQSeries process, XA function calls can be made to the external resource manager from multiple threads at the same time.

PROCESS
Means that the resource manager is not "thread safe". In a multithreaded MQSeries process, only one XA function call at a time can be made to the resource manager.

The `ThreadOfControl` entry does not apply to XA function calls issued by the queue manager in a multithreaded application process. In general, an application that has concurrent units of work on different threads requires this mode of operation to be supported by each of the resource managers.

The Channels stanza

The `Channels` stanza contains information about the channels.

MaxChannels=100|*number*
This attribute specifies the maximum number of channels allowed. The default is 100.

MaxActiveChannels=*MaxChannels_value*
This attribute specifies the maximum number of channels allowed to be active at any time. The default is the value specified on the `MaxChannels` attribute.

MaxInitiators=3|*number*
This attribute specifies the maximum number of initiators.

MQIBINDTYPE=FASTPATH|STANDARD
This attribute specifies the binding for applications.

FASTPATH
Channels connect using MQCONNX FASTPATH. That is, there is no agent process.

STANDARD
Channels connect using STANDARD.

AdoptNewMCA=NO|SVR|SDR|RCVR|CLUSRCVR|ALL|FASTPATH

If MQSeries receives a request to start a channel but finds that an amqcrsta process already exists for the same channel, the existing process must be stopped before the new one can start. The `AdoptNewMCA` attribute allows you to control the termination of an existing process and the startup of a new one for a specified channel type.

If you specify the `AdoptNewMCA` attribute for a given channel type but the new channel fails to start because the channel is already running:

1. The new channel tries to stop the previous one by politely inviting it to end.
2. If the previous channel server does not respond to this invitation by the time the AdoptNewMCATimeout wait interval expires, the process (or the thread) for the previous channel server is killed.
3. If the previous channel server has not ended after step 2, and after the AdoptNewMCATimeout wait interval expires for a second time, MQSeries ends the channel with a "CHANNEL IN USE" error.

You specify one or more values, separated by commas or blanks, from the following list:

NO	The `AdoptNewMCA` feature is not required. This is the default.
SVR	Adopt server channels
SDR	Adopt sender channels
RCVR	Adopt receiver channels
CLUSRCVR	Adopt cluster receiver channels
ALL	Adopt all channel types, except for FASTPATH channels
FASTPATH	Adopt the channel if it is a FASTPATH channel. This happens only if the appropriate channel type is also specified, for example, AdoptNewMCA=RCVR,SVR,FASTPATH

Attention!
The AdoptNewMCA attribute may behave in an unpredictable fashion with FASTPATH channels because of the internal design of the queue manager. Therefore exercise great caution when enabling the `AdoptNewMCA` attribute for FASTPATH channels.

AdoptNewMCATimeout=60|1—3600

This attribute specifies the amount of time, in seconds, that the new process should wait for the old process to end. Specify a value, in seconds, in the range 1—3600. The default value is 60.

AdoptNewMCACheck=QM|ADDRESS|NAME|ALL

The `AdoptNewMCACheck` attribute allows you to specify the type checking required when enabling the `AdoptNewMCA` attribute. It is important for you to perform all three of the following checks, if possible, to protect your channels from being, inadvertently or maliciously, shut down. At the very least check that the channel names match.

Specify one or more values, separated by commas or blanks, from the following:

QM This means that listener process should check that the queue manager names match.

ADDRESS
This means that the listener process should check the communications address. For example, the TCP/IP address.

NAME
This means that the listener process should check that the channel names match.

ALL You want the listener process to check for matching queue manager names, the communications address, and for matching channel names.

`AdoptNewMCACheck=NAME,ADDRESS` is the default for FAP1, FAP2, and FAP3, while `AdoptNewMCACheck=NAME,ADDRESS,QM` is the default for FAP4 and later.

The LU62 and TCP stanzas

These stanzas specify network protocol configuration parameters. They override the default attributes for channels.

Note: Only attributes representing changes to the default values need to be specified.

LU62
The following attributes can be specified:

TPName
This attribute specifies the TP name to start on the remote site.

LocalLU
This is the name of the logical unit to use on local systems.

TCP
The following attributes can be specified:

Port=1414|*port_number*
This attribute specifies the default port number, in decimal notation, for TCP/IP sessions. The "well known" port number for MQSeries is 1414.

KeepAlive=YES|NO
Use this attribute to switch the KeepAlive function on or off. KeepAlive=YES causes TCP/IP to check periodically that the other end of the connection is still available. If it is not, the channel is closed.

ListenerBacklog=number
When receiving on TCP/IP, a maximum number of outstanding connection requests is set. This can be considered to be a *backlog* of requests waiting on the TCP/IP port for the listener to accept the request. The default listener backlog values are shown in Table 8.

Table 8. Default outstanding connection requests (TCP)

Platform	Default ListenerBacklog value
OS/390	255
OS/2 Warp	10
Windows NT Server	100
Windows NT Workstation	5
AS/400	255

Table 8. Default outstanding connection requests (TCP) (continued)

Platform	Default ListenerBacklog value
Sun Solaris	100
HP-UX	20
AIX V4.2 or later	100
AIX V4.1 or earlier	10
All other platforms	5

If the backlog reaches the values shown in Table 8 on page 172, the TCP/IP connection is rejected and the channel will not be able to start.

For MCA channels, this results in the channel going into a RETRY state and retrying the connection at a later time.

For client connections, the client receives an MQRC_Q_MGR_NOT_AVAILABLE reason code from MQCONN and should retry the connection at a later time.

The `ListenerBacklog` attribute allows you to override the default number of outstanding requests for the TCP/IP listener.

Note: Some operating systems support a larger value than the default shown. If necessary, this can be used to avoid reaching the connection limit.

The ExitPath stanza

ExitDefaultPath=*string*
The ExitDefaultPath attribute specifies the location of:
- Channel exits for clients
- Channel exits and data conversion exits for servers

The exit path is read from the ClientExitPath stanza in the `mqs.ini` file for clients and from this (ExitPath) stanza for servers.

Example mqs.ini and qm.ini files

Figure 19 on page 174 shows an example of an `mqs.ini` file in MQSeries for Compaq OpenVMS.

Changing MQSeries configuration file

```
#*******************************************************************#
#* Module Name: mqs.ini                                            *#
#* Type       : MQSeries Configuration File                        *#
#* Function   : Define MQSeries resources for the node             *#
#*                                                                 *#
#*******************************************************************#
#* Notes      :                                                    *#
#* 1) This is an example MQSeries configuration file               *#
#*                                                                 *#
#*******************************************************************#
AllQueueManagers:
   #****************************************************************#
   #* The path to the qmgrs directory, below which queue manager data  *#
   #* is stored                                                    *#
   #****************************************************************#
   DefaultPrefix=mqs_root:[mqm]

ClientExitPath:
   ExitsDefaultPath=mqs_root:[mqm.exits]

LogDefaults:
   LogPrimaryFiles=3
   LogSecondaryFiles=2
   LogFilePages=1024
   LogType=CIRCULAR
   LogBufferPages=17
   LogDefaultPath=mqs_root:[mqm.log]
QueueManager:
   Name=saturn.queue.manager
   Prefix=mqs_root:[mqm]
   Directory=saturn$queue$manager
DefaultQueueManager:
   Name=saturn.queue.manager
QueueManager:
   Name=pluto.queue.manager
   Prefix=mqs_root:[mqm]
   Directory=pluto$queue$manager
```

Figure 19. Example of an MQSeries configuration file for MQSeries for Compaq OpenVMS systems

Figure 20 on page 175 shows how groups of attributes might be arranged in a queue manager configuration file in MQSeries for Compaq OpenVMS.

```
#*******************************************************************#
#* Module Name: qm.ini                                             *#
#* Type       : MQSeries queue manager configuration file          *#
#  Function   : Define the configuration of a single queue manager *#
#*                                                                 *#
#*******************************************************************#
#* Notes      :                                                    *#
#* 1) This file defines the configuration of the queue manager     *#
#*                                                                 *#
#*******************************************************************#
ExitPath:
   ExitsDefaultPath=mqm_root:[mqm.exits]

Service:
   Name=AuthorizationService
   EntryPoints=9

ServiceComponent:
   Service=AuthorizationService
   Name=MQSeries.UNIX.auth.service
   Module=amqzfu
   ComponentDataSize=0

Service:
   Name=NameService
   EntryPoints=5

ServiceComponent:
   Service=NameService
   Name=MQSeries.DCE.name.service
   Module=amqzfa
   ComponentDataSize=0

Log:
   LogPrimaryFiles=3
   LogSecondaryFiles=2
   LogFilePages=1024
   LogType=CIRCULAR
   LogBufferPages=17
   LogPath=mqm_root:[mqm.log.saturn$queue$manager]

XAResourceManager:
  Name=Oracle Resource Manager Bank
  SwitchFile=sys$share:oraswit0.exe
  XAOpenString=MQBankDB
  XACloseString=
  ThreadOfControl=PROCESS


CHANNELS:
  MaxChannels = 20         ; Maximum number of Channels allowed.
                           ; Default is 100.
  MaxActiveChannels = 10   ; Maximum number of Channels allowed to be
                           ; active at any time.  The default is the
                           ; value of MaxChannels.

TCP:                       ; TCP/IP entries.
  KeepAlive = Yes          ; Switch KeepAlive on
```

Figure 20. Example queue manager configuration file for MQSeries for Compaq OpenVMS

Notes:

MQSeries on the node is using the default locations for queue managers and for the logs.

Changing MQSeries configuration file

The queue manager saturn.queue.manager is the default queue manager for the node. The directory for files associated with this queue manager has been automatically transformed into a valid file name for the OpenVMS file system.

Because the MQSeries configuration file is used to locate the data associated with queue managers, a nonexistent or incorrect configuration file can cause some or all MQSeries commands to fail. Also, applications cannot connect to a queue manager that is not defined in the MQSeries configuration file.

Chapter 14. Problem determination

This chapter suggests some ways to deal with the problems you may have using MQSeries for Compaq OpenVMS.

Not all problems can be solved immediately; for example, performance problems may be caused by the limitations of your hardware. Also, if you think that the cause of the problem is in the MQSeries code, contact your IBM® Support Center. This chapter contains these sections:
- "Preliminary checks"
- "Common programming errors" on page 180
- "What to do next" on page 180
- "Application design considerations" on page 183
- "Incorrect output" on page 184
- "Error logs" on page 187
- "Dead-letter queues" on page 191
- "Configuration files and problem determination" on page 191
- "Using MQSeries trace" on page 191
- "First failure support technology (FFST)" on page 192
- "Problem determination with clients" on page 197

Preliminary checks

Problems with MQSeries typically arise from one of the following components:
- MQSeries
- The network
- The application
- The underlying operating system.

The sections that follow provide some basic questions that you should consider while investigating the problem.

Has MQSeries run successfully before?

If MQSeries has not run successfully before, it may not have been set up correctly. See *MQSeries for Compaq OpenVMS Alpha, Quick Beginnings, Version 5.1* to check that MQSeries has been installed and set up correctly.

Are there any error messages?

MQSeries uses error logs to capture messages concerning the operation of MQSeries itself, any queue managers that you start, and error data coming from the channels that are in use. Check the error logs to see if any messages have been recorded that are associated with your problem.

See "Error logs" on page 187 for information about the contents of the error logs, and their locations.

Are there any return codes explaining the problem?

If your application gets a return code indicating that a Message Queue Interface (MQI) call has failed, refer to the *MQSeries Application Programming Reference* book for a description of that return code.

Can you reproduce the problem?

If you can reproduce the problem, consider the conditions under which it is reproduced:

- Is it caused by a command or an equivalent administration request?

  Does the operation work if it is entered by another method? If the command works if it is entered on the command line, but not otherwise, check that the command server has not stopped, and that the queue definition of the SYSTEM.ADMIN.COMMAND.QUEUE has not been changed.
- Is it caused by a program? Does it fail on all MQSeries systems and all queue managers, or only on some?
- Can you identify any application that always seems to be running in the system when the problem occurs? If so, examine the application to see if it is in error.

Have any changes been made since the last successful run?

When you are considering changes that might recently have been made, think about the MQSeries system, and also about the other programs it interfaces with, the hardware, and any new applications. Consider also the possibility that a new application that you are not aware of might have been run on the system.

- Have you changed, added, or deleted any queue definitions?
- Have you changed or added any channel definitions? Changes may have been made to either MQSeries channel definitions or any underlying communications definitions required by your application.
- Do your applications deal with return codes that they might get as a result of any changes you have made?

Has the application run successfully before?

If the problem appears to involve one particular application, consider whether the application has run successfully before.

Before you answer **Yes** to this question, consider the following:

- Have any changes been made to the application since it last ran successfully?

  If so, it is likely that the error lies somewhere in the new or modified part of the application. Take a look at the changes and see if you can find an obvious reason for the problem. Is it possible to retry using a back level of the application?
- Have all the functions of the application been fully exercised before?

  Could it be that the problem occurred when part of the application that had never been invoked before was used for the first time? If so, it is likely that the error lies in that part of the application. Try to find out what the application was doing when it failed, and check the source code in that part of the program for errors.

  If a program has been run successfully on many previous occasions, check the current queue status, and the files that were being processed when the error occurred. It is possible that they contain some unusual data value that causes a rarely used path in the program to be invoked.
- Does the application check all return codes?

  Has your MQSeries system been changed, perhaps in a minor way, such that your application does not check the return codes it receives as a result of the change. For example, does your application assume that the queues it accesses can be shared? If a queue has been redefined as exclusive, can your application deal with return codes indicating that it can no longer access that queue?

- Does the application run on other MQSeries systems?

  Could it be that there is something different about the way that this MQSeries system is set up which is causing the problem? For example, have the queues been defined with the same message length or priority?

If the application has not run successfully before

If your application has not yet run successfully, you need to examine it carefully to see if you can find any errors.

Before you look at the code, and depending upon which programming language the code is written in, examine the output from the translator, or the compiler and linkage editor, if applicable, to see if any errors have been reported.

If your application fails to translate, compile, or link-edit into the load library, it will also fail to run if you attempt to invoke it. See the *MQSeries Application Programming Reference* book for information about building your application.

If the documentation shows that each of these steps was accomplished without error, you should consider the coding logic of the application. Do the symptoms of the problem indicate the function that is failing and, therefore, the piece of code in error? See “Common programming errors” on page 180 for some examples of common errors that cause problems with MQSeries applications.

Does the problem affect specific parts of the network?

You might be able to identify specific parts of the network that are affected by the problem (remote queues, for example). If the link to a remote message queue manager is not working, the messages cannot flow to a remote queue.

Check that the connection between the two systems is available, and that the intercommunication component of MQSeries has been started.

Check that messages are reaching the transmission queue, and check the local queue definition of the transmission queue and any remote queues.

Have you made any network-related changes, or changed any MQSeries definitions, that might account for the problem?

Does the problem occur at specific times of the day?

If the problem occurs at specific times of day, it could be that it is dependent on system loading. Typically, peak system loading is at mid-morning and mid-afternoon, so these are the times when load-dependent problems are most likely to occur. (If your MQSeries network extends across more than one time zone, peak system loading might seem to occur at some other time of day.)

Is the problem intermittent?

An intermittent problem could be caused by failing to take into account the fact that processes can run independently of each other. For example, a program may issue an MQGET call, without specifying a wait option, before an earlier process has completed. An intermittent problem may also be seen if your application tries to get a message from a queue while the call that put the message is in-doubt (that is, before it has been committed or backed out).

Have you applied any service updates?

If a service update has been applied to MQSeries, check that the update action completed successfully and that no error message was produced.

- Did the update have any special instructions?
- Was any test run to verify that the update had been applied correctly and completely?
- Does the problem still exist if MQSeries is restored to the previous service level?
- If the installation was successful, check with the IBM Support Center for any patch error.
- If a patch has been applied to any other program, consider the effect it might have on the way MQSeries interfaces with it.

Do you need to apply an updates?

MQSeries depends upon the underlying operating system (OpenVMS) and various networking products, such as TCP/IP. Check with the appropriate vendor to ensure that you have applied all necessary service updates for these products.

Common programming errors

The errors in the following list illustrate the most common causes of problems encountered while running MQSeries programs. You should consider the possibility that the problem with your MQSeries system could be caused by one or more of these errors:

- Assuming that queues can be shared, when they are in fact exclusive.
- Passing incorrect parameters in an MQI call.
- Passing insufficient parameters in an MQI call. This may mean that MQI cannot set up completion and reason codes for your application to process.
- Failing to check return codes from MQI requests.
- Passing variables with incorrect lengths specified.
- Passing parameters in the wrong order.
- Failing to initialize *MsgId* and *CorrelId* correctly.

What to do next

Perhaps the preliminary checks have enabled you to find the cause of the problem. If so, you should now be able to resolve it, possibly with the help of other books in the MQSeries library (see the Bibliography) and in the libraries of other licensed programs.

If you have not yet found the cause, you must start to look at the problem in greater detail.

The purpose of this section is to help you identify the cause of your problem if the preliminary checks have not enabled you to find it.

When you have established that no changes have been made to your system, and that there are no problems with your application programs, choose the option that best describes the symptoms of your problem.

- "Have you obtained incorrect output?" on page 181
- "Have you failed to receive a response from a PCF command?" on page 181
- "Does the problem affect only remote queues?" on page 182

If none of these symptoms describe your problem, consider whether it might have been caused by another component of your system.

Have you obtained incorrect output?

In this book, “incorrect output” refers to your application:

- Not receiving a message that it was expecting.
- Receiving a message containing unexpected or corrupted information.
- Receiving a message that it was not expecting, for example, one that was destined for a different application.

In all cases, check that any queue or queue manager aliases that your applications are using are correctly specified and accommodate any changes that have been made to your network.

If an MQSeries error message is generated, all of which are prefixed with the letters “AMQ”, you should look in the error log. See “Error logs” on page 187 for further information.

Have you failed to receive a response from a PCF command?

If you have issued a command but you have not received a response, consider the following questions:

- Is the command server running?

  Work with the **dspmqcsv** command to check the status of the command server.
  - If the response to this command indicates that the command server is not running, use the **strmqcsv** command to start it.
  - If the response to the command indicates that the SYSTEM.ADMIN.COMMAND.QUEUE is not enabled for MQGET requests, enable the queue for MQGET requests.
- Has a reply been sent to the dead-letter queue?

  The dead-letter queue header structure contains a reason or feedback code describing the problem. See the *MQSeries Application Programming Reference* book for information about the dead-letter queue header structure (MQDLH).

  If the dead-letter queue contains messages, you can use the provided browse sample application (amqsbcg) to browse the messages using the MQGET call. The sample application steps through all the messages on a named queue for a named queue manager, displaying both the message descriptor and the message context fields for all the messages on the named queue.
- Has a message been sent to the error log?

  See “Error logs” on page 187 for further information.
- Are the queues enabled for put and get operations?
- Is the `WaitInterval` long enough?

  If your MQGET call has timed out, a completion code of MQCC_FAILED and a reason code of MQRC_NO_MSG_AVAILABLE are returned. (See the *MQSeries Application Programming Reference* book for information about the `WaitInterval` field, and completion and reason codes from MQGET.)
- If you are using your own application program to put commands onto the SYSTEM.ADMIN.COMMAND.QUEUE, do you need to take a syncpoint?

  Unless you have specifically excluded your request message from syncpoint, you need to take a syncpoint before attempting to receive reply messages.

- Are the MAXDEPTH and MAXMSGL attributes of your queues set sufficiently high?
- Are you using the `CorrelId` and `MsgId` fields correctly?

  Set the values of `MsgId` and `CorrelId` in your application to ensure that you receive all messages from the queue.

Try stopping the command server and then restarting it, responding to any error messages that are produced.

If the system still does not respond, the problem could be with either a queue manager or the whole of the MQSeries system. First try stopping individual queue managers to try and isolate a failing queue manager. If this does not reveal the problem, try stopping and restarting MQSeries, responding to any messages that are produced in the error log.

If the problem still occurs after restart, contact your IBM Support Center for help.

Are some of your queues failing?

If you suspect that the problem occurs with only a subset of queues, check the local queues that you think are having problems:

1. Display the information about each queue. You can use the MQSC command DISPLAY QUEUE to display the information.
2. Use the data displayed to do the following checks:
   - If CURDEPTH is at MAXDEPTH, this indicates that the queue is not being processed. Check that all applications are running normally.
   - If CURDEPTH is not at MAXDEPTH, check the following queue attributes to ensure that they are correct:
     – If triggering is being used:
       - Is the trigger monitor running?
       - Is the trigger depth too great? That is, does it generate a trigger event often enough?
       - Is the process name correct?
       - Is the process available and operational?

     – Can the queue be shared? If not, another application could already have it open for input.

     – Is the queue enabled appropriately for GET and PUT?
   - If there are no application processes getting messages from the queue, determine why this is so. It could be because the applications need to be started, a connection has been disrupted, or the MQOPEN call has failed for some reason.

     Check the queue attributes IPPROCS and OPPROCS. These attributes indicate whether the queue has been opened for input and output. If a value is zero, it indicates that no operations of that type can occur. Note that the values may have changed and that the queue was open but is now closed.

     You need to check the status at the time you expect to put or get a message.

If you are unable to solve the problem, contact your IBM Support Center for help.

Does the problem affect only remote queues?

If the problem affects only remote queues, check the following:

- Check that required channels have been started and are triggerable, and that any required initiators are running.

- Check that the programs that should be putting messages to the remote queues have not reported problems.
- If you use triggering to start the distributed queuing process, check that the transmission queue has triggering set on. Also, check that the channel initiator is running.
- Check the error logs for messages indicating channel errors or problems.
- If necessary, start the channel manually. See the *MQSeries Intercommunication* book for information about how to do this.

See the *MQSeries Intercommunication* book for information about how to define channels.

Application design considerations

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

For more information about application design, see the *MQSeries Application Programming Guide*.

Effect of message length

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

Effect of message persistence

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

Searching for a particular message

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

Queues that contain messages of different lengths

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

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

Frequency of syncpoints

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

Use of the MQPUT1 call

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

Incorrect output

The term “incorrect output” can be interpreted in many different ways. For the purpose of problem determination within this book, the meaning is explained in “Have you obtained incorrect output?” on page 181.

Two types of incorrect output are discussed in this section:
- Messages that do not appear when you are expecting them
- Messages that contain the wrong information, or information that has been corrupted

Additional problems that you might find if your application includes the use of distributed queues are also discussed.

Messages that do not appear on the queue

If messages do not appear when you are expecting them, check for the following:
- Has the message been put on the queue successfully?
  - Has the queue been defined correctly. For example, is MAXMSGL sufficiently large?
  - Is the queue enabled for putting?
  - Is the queue already full? This could mean that an application was unable to put the required message on the queue.
- Are you able to get any messages from the queue?
  - Do you need to take a syncpoint?

    If messages are being put or retrieved within syncpoint, they are not available to other tasks until the unit of recovery has been committed.
  - Is your wait interval long enough?

    You can set the wait interval as an option for the MQGET call. You should ensure that you are waiting long enough for a response.
  - Are you waiting for a specific message that is identified by a message or correlation identifier (*MsgId* or *CorrelId*)?

  Check that you are waiting for a message with the correct `MsgId` or `CorrelId`. A successful MQGET call sets both these values to that of the message retrieved, so you may need to reset these values in order to get another message successfully.

  Also, check whether you can get other messages from the queue.
  - Can other applications get messages from the queue?
  - Was the message you are expecting defined as persistent?

    If not, and MQSeries has been restarted, the message has been lost.
  - Has another application got exclusive access to the queue?

If you are unable to find anything wrong with the queue, and MQSeries is running, make the following checks on the process that you expected to put the message on to the queue:

- Did the application get started?

  If it should have been triggered, check that the correct trigger options were specified.
- Did the application stop?
- Is a trigger monitor running?
- Was the trigger process defined correctly?
- Did the application complete correctly?

  Look for evidence of an abnormal end in the job log.
- Did the application commit its changes, or were they backed out?

If multiple transactions are serving the queue, they can conflict with one another. For example, suppose one transaction issues an MQGET call with a buffer length of zero to find out the length of the message, and then issues a specific MQGET call specifying the `MsgId` of that message. However, in the meantime, another transaction issues a successful MQGET call for that message, so the first application receives a reason code of MQRC_NO_MSG_AVAILABLE. Applications that are expected to run in a multi-server environment must be designed to cope with this situation.

Consider that the message could have been received, but that your application failed to process it in some way. For example, did an error in the expected format of the message cause your program to reject it? If this is the case, refer to "Messages that contain unexpected or corrupted information".

Messages that contain unexpected or corrupted information

If the information contained in the message is not what your application was expecting, or has been corrupted in some way, consider the following points:

- Has your application, or the application that put the message onto the queue, changed?

  Ensure that all changes are simultaneously reflected on all systems that need to be aware of the change.

  For example, the format of the message data may have been changed, in which case, both applications must be recompiled to pick up the changes. If one application has not been recompiled, the data will appear corrupt to the other.
- Is an application sending messages to the wrong queue?

Incorrect output

Check that the messages your application is receiving are not really intended for an application servicing a different queue. If necessary, change your security definitions to prevent unauthorized applications from putting messages on to the wrong queues.

If your application has used an alias queue, check that the alias points to the correct queue.

- Has the trigger information been specified correctly for this queue?

  Check that your application should have been started; or should a different application have been started?

If these checks do not enable you to solve the problem, you should check your application logic, both for the program sending the message, and for the program receiving it.

Problems with incorrect output when using distributed queues

If your application uses distributed queues, you should also consider the following points:

- Has MQSeries been correctly installed on both the sending and receiving systems, and correctly configured for distributed queuing?
- Are the links available between the two systems?

  Check that both systems are available, and connected to MQSeries. Check that the connection between the two systems, and the channels between the two queue managers, are active.
- Is triggering set on in the sending system?
- Is the message you are waiting for a reply message from a remote system?

  Check that triggering is activated in the remote system.
- Is the queue already full?

  This could mean that an application was unable to put the required message onto the queue. If this is so, check if the message has been put onto the dead-letter queue.

  The dead-letter queue header contains a reason or feedback code explaining why the message could not be put onto the target queue. See the *MQSeries Application Programming Reference* book for information about the dead-letter queue header structure.
- Is there a mismatch between the sending and receiving queue managers?

  For example, the message length could be longer than the receiving queue manager can handle.
- Are the channel definitions of the sending and receiving channels compatible?

  For example, a mismatch in sequence number wrap stops the distributed queuing component. See the *MQSeries Intercommunication* book for more information about distributed queuing.
- Is data conversion involved? If the data formats between the sending and receiving applications differ, data conversion is necessary. Automatic conversion occurs when the MQGET is issued if the format is recognized as one of the built-in formats.

  If the data set is not recognized for conversion, the data conversion exit is taken to allow you to perform the translation with your own routines.

  An exception to the above occurs if you are sending data to MQSeries for MVS/ESA.

Refer to the *MQSeries Intercommunication* book for further details of data conversion.

Error logs

MQSeries uses a number of error logs to capture messages concerning the operation of MQSeries itself, any queue managers that you start, and error data coming from the channels that are in use.

The location of the error logs depends on whether the queue manager name is known and whether the error is associated with a client.

- If the queue manager name is known and the queue manager is available:

  `MQS_ROOT:[MQM.QMGRS.QMgrName.ERRORS]AMQERR01.LOG`
- If the queue manager is not available:

  `MQS_ROOT:[MQM.QMGRS.$SYSTEM.ERRORS]AMQERR01.LOG`
- If an error has occurred with a client application:

  `MQS_ROOT:[MQM.ERRORS]AMQERR01.LOG`
- First Failure Support Technology® (FFST) – see "How to examine the FFSTs" on page 192.

Note: In the case of clients, the errors are stored on the client's root drive.

Log files

At installation time an `[MQM.QMGRS.$SYSTEM.ERRORS]` directory is created in the QMGRS file path. The errors subdirectory can contain up to three error log files named:

- AMQERR01.LOG
- AMQERR02.LOG
- AMQERR03.LOG

After you have created a queue manager, three error log files are created when they are needed by the queue manager. These files have the same names as the $SYSTEM ones, that is AMQERR01, AMQERR02, and AMQERR03, and each has a capacity of 256 KB. The files are placed in the errors subdirectory of each queue manager that you create.

As error messages are generated they are placed in AMQERR01. When AMQERR01 gets bigger than 256 KB it is copied to AMQERR02. Before the copy, AMQERR02 is copied to AMQERR03.LOG. The previous contents, if any, of AMQERR03 are discarded.

The latest error messages are thus always placed in AMQERR01, the other files being used to maintain a history of error messages.

All messages relating to channels are also placed in the appropriate queue manager's errors files unless the name of their queue manager is unknown or the queue manager is unavailable. When the queue manager name is unavailable or its name cannot be determined, channel-related messages are placed in the [MQM.QMGRS.$SYSTEM.ERRORS] subdirectory.

To examine the contents of any error log file, use your usual OpenVMS editor.

Early errors

There are a number of special cases where the above error logs have not yet been established and an error occurs. MQSeries attempts to record any such errors in an error log. The location of the log depends on how much of a queue manager has been established.

If, due to a corrupt configuration file for example, no location information can be determined, errors are logged to an errors directory that is created at installation time on the root directory, mqm.

If the MQSeries configuration file is readable, and the DefaultPrefix attribute of the AllQueueManagers stanza is readable, errors are logged in the DefaultPrefix[.errors] directory.

For further information about configuration files, see "Chapter 13. Configuring MQSeries" on page 159.

Operator messages

In MQSeries for Compaq OpenVMS, operator messages identify normal errors, typically caused directly by users doing things like using parameters that are not valid on a command. Operator messages are national language (NLS) enabled, with message catalogs installed in standard locations.

These messages are written to the associated window, if any, and are also written to the error log AMQERR01.LOG in the queue manager directory. For example:

```
MQS_ROOT:[MQM.QMGRS.QUEUE$MANAGER.ERRORS]
```

Some errors are logged to the AMQERR01.LOG file in the queue manager directory and others to the $SYSTEM directory copy of the error log.

Example error log

This example shows part of a MQSeries for Compaq OpenVMS error log:

```
 ...
06/29/00  09:41:39 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this may
be required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive
medium to release space in the log directory. If you move any of the log
files required to recreate objects from their media images, you will
have to restore them to recreate the objects.
---------------------------------------------                        --
06/29/00  09:41:39 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images. Any
log files prior to this will not be accessed by media recovery operations.
ACTION: You can move log files older than S0000000.LOG to an archive
medium to release space in the log directory.
---------------------------------------------                        --
06/29/00  09:42:05 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this
may be required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive
medium to release space in the log directory. If you move any of the log
files required to recreate objects from their media images,
you will have to restore them to recreate the objects.
```

```
---------------------------------------                                  --
06/29/00  09:42:05 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images.  Any
log files prior to this will not be accessed by media recovery operations.
ACTION: You can move log files older than S0000000.LOG to an archive
medium to release space in the log directory.
---------------------------------------                                  --
06/29/00  09:42:06 AMQ8003: MQSeries queue manager started.

EXPLANATION: MQSeries queue manager BKM1 started.
ACTION: None.
---------------------------------------                                  --
06/29/00  09:42:06 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this
may be required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive
medium to release space in the log directory. If you move any of the
log files required to recreate objects from their media images,
you will have to restore them to recreate the objects.
---------------------------------------                                  --
06/29/00  09:42:06 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images.
Any log files prior to this will not be accessed by media recovery
operations.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory.
---------------------------------------                                  --
06/29/00  09:46:27 AMQ7030: Request to quiesce the queue manager accepted.
The queue manager will stop when there is no further work for it to
perform.

EXPLANATION: You have requested that the queue manager end when there is no
more work for it.  In the meantime, it will refuse new applications
that attempt to start, although it allows those already running to
complete their work.
ACTION: None.
---------------------------------------                                  --
06/29/00  09:46:43 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this may be
required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive
medium to release space in the log directory. If you move any of the
log files required to recreate objects from their media images, you
will have to restore them to recreate the objects.
-----------------------------------                                      --
06/29/00  09:46:43 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images. Any
log files prior to this will not be accessed by media recovery operations.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory.
-----------------------------------                                      --
06/29/00  09:46:44 AMQ8004: MQSeries queue manager ended.

EXPLANATION: MQSeries queue manager BKM1 ended.
ACTION: None.
-----------------------------------                                      --
06/29/00  09:46:59 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this
may be required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory. If you move any of the log files
required to recreate objects from their media images, you will have to
```

```
restore them to recreate the objects.
-------------------------------------                                    --
06/29/00  09:47:00 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images. Any log
files prior to this will not be accessed by media recovery operations.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory.
-------------------------------------                                    --
06/29/00  09:47:08 AMQ7472: Object TEST1, type queue damaged.

EXPLANATION: Object TEST1, type queue has been marked as damaged. This
indicates that the queue manager was either unable to access the object in
the file system, or that some kind of inconsistency with the data in
the object was detected.
ACTION: If a damaged object is detected, the action performed depends on
whether the queue manager supports media recovery and when the damage
was detected. If the queue manager does not support media recovery,
you must delete the object as no recovery is possible. If the queue manager
does support media recovery and the damage is detected during the processing
performed when the queue manager is being started, the queue manager will
automatically initiate media recovery of the object.  If the queue
manager supports media recovery and the damage is detected once the queue
manager has started, it may be recovered from a media image using the
rcrmqobj command or it may be deleted.
--------------------------------------                                   --
06/29/00  09:47:09 AMQ8003: MQSeries queue manager started.

EXPLANATION: MQSeries queue manager BKM1 started.
ACTION: None.
---------------------------------------                                  --
06/29/00  09:47:09 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this may be
required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory. If you move any of the log files
required to recreate objects from their media images, you will have to
restore them to recreate the objects.
------------------------------------                                     --
06/29/00  09:47:10 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images. Any log
files prior to this will not be accessed by media recovery operations.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory.
-------------------------------------                                    --
06/29/00  09:47:47 AMQ7081: Object TEST1, type queue recreated.

EXPLANATION: The object TEST1, type queue was recreated from its media
image.
ACTION: None.
-------------------------------------                                    --
06/29/00  11:22:10 AMQ7467: The oldest log file required to start queue
manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to restart the queue manager. Log records older than this may
be required for media recovery.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory. If you move any of the log files
required to recreate objects from their media images, you will have
to restore them to recreate the objects.
-----------------------------------                                      --
06/29/00  11:22:10 AMQ7468: The oldest log file required to perform media
recovery of queue manager BKM1 is S0000000.LOG.

EXPLANATION: The log file S0000000.LOG contains the oldest log record
required to recreate any of the objects from their media images. Any log files
prior to this will not be accessed by media recovery operations.
ACTION: You can move log files older than S0000000.LOG to an archive medium
to release space in the log directory.
--------------------------------                                         --
06/29/00  11:22:11 AMQ8004: MQSeries queue manager ended.
```

```
EXPLANATION: MQSeries queue manager BKM1 ended.
ACTION: None.
-------------------------------                                        --
 ...
```

Dead-letter queues

Messages that cannot be delivered for some reason are placed on the dead-letter queue. You can check whether the queue contains any messages by issuing an MQSC DISPLAY QUEUE command. If the queue contains messages, you can use the provided browse sample application (amqsbcg) to browse messages on the queue using the MQGET call. The sample application steps through all the messages on a named queue for a named queue manager, displaying both the message descriptor and the message context fields for all the messages on the named queue.

You must decide how to dispose of any messages found on the dead-letter queue, depending on the reasons for the messages being put on the queue.

Problems may occur if you do not have a dead-letter queue on each queue manager you are using. When the queue manager is created using the **crtmqm** command, a dead-letter queue called SYSTEM.DEAD.LETTER.QUEUE is automatically created as a default object. However, this queue is not defined as the dead-letter queue for the queue manager. See “Defining a dead-letter queue” on page 41.

Configuration files and problem determination

Configuration file errors typically prevent queue managers from being found and result in “queue manager unavailable” type errors.

There are several checks you can make on the configuration files:
- Ensure that the configuration files exist.
- Ensure that they have appropriate permissions, for example:
  ```
        MQS.INI;1   MQM    (RWED, RWED, RW, R)
  (identifier=MQM, ACCESS=READ+WRITE+EXECUTE+DELETE+CONTROL)
  ```
- Ensure that the MQSeries configuration file references the correct queue manager and log directories.

Using MQSeries trace

MQSeries for Compaq OpenVMS uses the following commands for the trace facility:
- **strmqtrc** – see “strmqtrc (Start MQSeries trace)” on page 291
- **dspmqtrc** – see “dspmqtrc (Display MQSeries formatted trace output)” on page 246
- **endmqtrc** – see “endmqtrc (End MQSeries trace)” on page 257

The trace facility uses one file for each entity being traced, with the trace information being recorded in the appropriate file.

Files associated with trace are created in the directory MQS_ROOT:[MQM.TRACE].

The files in this directory include details of queue managers, as well as all early tracing and all $SYSTEM tracing.

Trace file names

Trace file names are constructed in the following way:

`AMQpppppppp.TRC`

where *pppppppp* is the process identifier (PID) of the process producing the trace.

Notes:

1. In MQSeries for Compaq OpenVMS, the value of the process identifier is always eight characters long.
2. There will be one trace file for each process running as part of the entity being traced.

Sample trace data

The following sample is an extract from an OpenVMS trace:

```
 ...
ID     ELAPSED_MSEC     DELTA_MSEC      APPL    SYSCALL KERNEL  INTERRUPT

30d     0 0 MQS CEI Exit!. 12484.1 xcsWaitEventSem rc=10806020
30d     0 0 MQS CEI Exit! 12484.1 zcpReceiveOnLink rc=20805311
30d     0 0 MQS FNC Entry 12484.1 zxcProcessChildren
30d     0 0 MQS CEI Entry. 12484.1 xcsRequestMutexSem
30d     1 0 MQS CEI Entry.. 12484.1 xcsHSHMEMBtoPTR
30d     1 0 MQS CEI Exit... 12484.1 xcsHSHMEMBtoPTR rc=00000000
30d     1 0 MQS FNC Entry.. 12484.1 xllSemGetVal
30d     1 0 MQS FNC Exit... 12484.1 xllSemGetVal rc=00000000
30d     1 0 MQS FNC Entry.. 12484.1 xllSemReq
30d     1 0 MQS FNC Exit... 12484.1 xllSemReq rc=00000000
30d     1 0 MQS CEI Exit.. 12484.1 xcsRequestMutexSem rc=00000000
30d     2 0 MQS CEI Entry. 12484.1 xcsReleaseMutexSem
30d     2 0 MQS CEI Entry.. 12484.1 xcsHSHMEMBtoPTR
30d     2 0 MQS CEI Exit... 12484.1 xcsHSHMEMBtoPTR rc=00000000
30d     2 0 MQS FNC Entry.. 12484.1 xllSemRel
30d     2 0 MQS FNC Exit... 12484.1 xllSemRel rc=00000000
30d     2 0 MQS CEI Exit.. 12484.1 xcsReleaseMutexSem rc=00000000
30d     2 0 MQS CEI Entry. 12484.1 xcsHSHMEMBtoPTR
 ...
```

Figure 21. Sample MQSeries for Compaq OpenVMS trace

Notes:

1. In this example the data is truncated. In a real trace, the complete function names and return codes are present.
2. The return codes are given as values, not literals.

First failure support technology (FFST)

Information that is normally recorded in FFST logs is, on MQSeries for Compaq OpenVMS, recorded in a file in the MQS_ROOT:[MQM.ERRORS] directory.

These errors are normally severe, unrecoverable errors and indicate either a configuration problem with the system or an MQSeries internal error.

How to examine the FFSTs

The files are named `AMQnnnnnnnn_mm.FDC`, where:

nnnnnnnn
: Is the process id reporting the error

mm Is a sequence number, normally 0

When a process creates an FFST it also writes an entry in the system error log. The record contains the name of the FFST file to assist in automatic problem tracking.

```
MQSeries First Failure Symptom Report
=====================================

Date/Time         :- Monday January 29 21:32:03 GMT 2001
Host Name         :- CELERY (Unknown)
PIDS              :- 5697175
LVLS              :- 510
Product Long Name :- MQSeries for OpenVMS Alpha
Vendor            :- IBM
Probe Id          :- ZX005025
Application Name  :- MQM
Component         :- zxcProcessChildren
Build Date        :- Jan  8 2001
Userid            :- [400,400] (SJACKSON)
Program Name      :- AMQZXMA0.EXE
Process           :- 202001DA
Thread            :- 00000001
QueueManager      :- JJJH
Major Errorcode   :- zrcX_PROCESS_MISSING
Minor Errorcode   :- OK
Probe Type        :- MSGAMQ5008
Probe Severity    :- 2
Probe Description :- AMQ5008: An essential MQSeries process 538968541
  cannot be found and is assumed to be terminated.
Arith1            :- 538968541 202001dd
VMS Errorcode     :- -SYSTEM-W-NONEXPR, nonexistent process (000008E8)

JPI Quota information:
======================
ASTCNT=247/250(98%) *                   BIOCNT=500/500(100%) *
BYTCNT=183616/183616(100%) *            DIOCNT=250/250(100%) *
ENQCNT=4885/5000(97%) *                 FILCNT=241/250(96%) *
PAGFILCNT=975280/1000000(97%) *         TQCNT=246/250(98%) *
FREPTECNT=2147483647                    APTCNT=0
GPGCNT=5808                             PPGCNT=5872
VIRTPEAK=203264                         DFWSCNT=1392
WSAUTH=2784                             WSAUTHEXT=65536
WSEXTENT=65536                          WSPEAK=11680
WSQUOTA=2784                            WSSIZE=15792
CPULIM=0                                MAXDETACH=0
MAXJOBS=0                               JOBPRCCNT=2
PAGEFLTS=2895                           PRCCNT=2/100(2%) +
(*) - % resource remaining, (+) - % resource used

Privilege and rights information:
=================================
CURPRIV=bugchk detach netmbx prmgbl sysgbl sysprv tmpmbx world
IMAGPRIV=bugchk prmgbl sysgbl world
AUTHPRIV=bugchk detach netmbx prmgbl sysgbl sysprv tmpmbx world
SJACKSON                                INTERACT
REMOTE                                  MQM
SYS
IMAGE_RIGHTS=
SYS$NODE_CELERY

SYI information:
================
ACTIVE CPU=1/1(100%) +                  CLUSTER NODES=1
FREE_GBLPAGES=16000528/16174643(98%) *  GBLPAGFIL=1000000
FREE_GBLSECTS=936/1550(60%) *           MEMSIZE=16384
PAGEFILE_FREE=16888/16888(100%) *       PAGE_SIZE=8192
SWAPFILE_FREE=936/936(100%) *           MAXPROCESSCNT=102
PROCSECTCNT=64                          BALSETCNT=100
WSMAX=65536                             NPAGEDYN=2269184
NPAGEVIR=9437184                        PAGEDYN=1597440
VIRTUALPAGECNT=2147483647               LOCKIDTBL_MAX=109437
PQL_DASTLM=24                           PQL_MASTLM=100
PQL_DBIOLM=32                           PQL_MBIOLM=100
PQL_DBYTLM=65536                        PQL_MBYTLM=100000
PQL_DCPULM=0                            PQL_MCPULM=0
PQL_DDIOLM=32                           PQL_MDIOLM=100
PQL_DFILLM=128                          PQL_MFILLM=100
```

```
| PQL_DPGFLQUOTA=65536                     PQL_MPGFLQUOTA=32768               |
| PQL_DPRCLM=32                            PQL_MPRCLM=10                      |
| PQL_DTQELM=16                            PQL_MTQELM=0                       |
| PQL_DWSDEFAULT=1392                      PQL_MWSDEFAULT=1392                |
| PQL_DWSQUOTA=2784                        PQL_MWSQUOTA=2784                  |
| PQL_DWSEXTENT=65536                      PQL_MWSEXTENT=65536                |
| PQL_DENQLM=128                           PQL_MENQLM=300                     |
| PQL_DJTQUOTA=4096                        PQL_MJTQUOTA=0                     |
| CLISYMTBL=750                            DEFMBXMXMSG=256                    |
| DEFMBXBUFQUO=1056                        CHANNELCNT=5000                    |
| DLCKEXTRASTK=2560                        PIOPAGES=575                       |
| CTLPAGES=256                             CTLIMGLIM=35                       |
| (*) - % resource remaining, (+) - % resource used                           |
|                                                                             |
+-----------------------------------------------------------------------------+

MQM Function Stack
zxcProcessChildren
xcsFFST

MQM Trace History
                     --> xllFreeSem
                     <-- xllFreeSem rc=OK
                     --> xcsFreeQuickCell
                      --> xllSpinLockRequest
                      <-- xllSpinLockRequest rc=OK
                      --> xstFreeCell
                      <-- xstFreeCell rc=OK
                      --> xllSpinLockRelease
                      <-- xllSpinLockRelease rc=OK
                     <-- xcsFreeQuickCell rc=OK
                    <-- xcsCloseEventSem rc=OK
                    --> xcsFreeMemBlock
                     --> xstFreeMemBlock
                      --> xcsRequestThreadMutexSem
                      <-- xcsRequestThreadMutexSem rc=OK
                      --> xcsReleaseThreadMutexSem
                      <-- xcsReleaseThreadMutexSem rc=OK
                      --> xstFreeBlockFromSharedMemSet
                       --> xllSpinLockSlowRequest
                       <-- xllSpinLockSlowRequest rc=OK
                       --> xllSpinLockRelease
                       <-- xllSpinLockRelease rc=OK
                       --> xstFreeBlockInExtent
                        --> xcsQueryMutexSem
                        <-- xcsQueryMutexSem rc=OK
                        --> xcsRequestMutexSem
                         --> xllSemReq
                          --> vms_mtx
                           --> vms_get_lock
                           <-- vms_get_lock rc=OK
                          <-- vms_mtx rc=OK
                         <-- xllSemReq rc=OK
                        <-- xcsRequestMutexSem rc=OK
                        --> xclDeleteMutexMem
                         --> xllCSCloseMutex
                          --> xihHANDLEtoSUBPOOLFn
                           --> xihGetConnSPDetailsFromList
                            --> xihGetConnSPDetails
                            <-- xihGetConnSPDetails rc=OK
                           <-- xihGetConnSPDetailsFromList rc=OK
                          <-- xihHANDLEtoSUBPOOLFn rc=OK
                          --> xllSpinLockSlowRequest
                          <-- xllSpinLockSlowRequest rc=OK
                          --> xllSpinLockRelease
                          <-- xllSpinLockRelease rc=OK
                          --> xllFreeSem
                          <-- xllFreeSem rc=OK
                          --> vms_mtx
                           --> vms_get_lock
                           <-- vms_get_lock rc=OK
                          <-- vms_mtx rc=OK
                          --> xcsFreeQuickCell
                           --> xllSpinLockRequest
                           <-- xllSpinLockRequest rc=OK
                           --> xstFreeCell
                           <-- xstFreeCell rc=OK
                           --> xllSpinLockRelease
                           <-- xllSpinLockRelease rc=OK
                          <-- xcsFreeQuickCell rc=OK
```

```
      <-- xllCSCloseMutex rc=OK
     <-- xclDeleteMutexMem rc=OK
     --> xstSerialiseExtent
      --> xllSpinLockRequest
      <-- xllSpinLockRequest rc=OK
     <-- xstSerialiseExtent rc=OK
     --> xstFreeChunk
      --> xstDeleteChunk
      <-- xstDeleteChunk rc=OK
      --> xstInsertChunk
      <-- xstInsertChunk rc=OK
     <-- xstFreeChunk rc=OK
     --> xstReleaseSerialisationOnExtent
      --> xllSpinLockRelease
      <-- xllSpinLockRelease rc=OK
     <-- xstReleaseSerialisationOnExtent rc=OK
    <-- xstFreeBlockInExtent rc=OK
   <-- xstFreeBlockFromSharedMemSet rc=OK
  <-- xstFreeMemBlock rc=OK
 <-- xcsFreeMemBlock rc=OK
<-- zcpDeleteIPC rc=OK
--> xcsReleaseMutexSem
 --> xllSemRel
  --> vms_mtx
   --> vms_get_lock
   <-- vms_get_lock rc=OK
  <-- vms_mtx rc=OK
 <-- xllSemRel rc=OK
<-- xcsReleaseMutexSem rc=OK
--> xcsFreeMemBlock
 --> xstFreeMemBlock
  --> xcsRequestThreadMutexSem
  <-- xcsRequestThreadMutexSem rc=OK
  --> xcsReleaseThreadMutexSem
  <-- xcsReleaseThreadMutexSem rc=OK
  --> xstFreeBlockFromSharedMemSet
   --> xllSpinLockSlowRequest
   <-- xllSpinLockSlowRequest rc=OK
   --> xllSpinLockRelease
   <-- xllSpinLockRelease rc=OK
   --> xstFreeBlockInExtent
    --> xcsQueryMutexSem
    <-- xcsQueryMutexSem rc=OK
    --> xcsRequestMutexSem
     --> xllSemReq
      --> vms_mtx
       --> vms_get_lock
       <-- vms_get_lock rc=OK
      <-- vms_mtx rc=OK
     <-- xllSemReq rc=OK
    <-- xcsRequestMutexSem rc=OK
    --> xclDeleteMutexMem
     --> xllCSCloseMutex
      --> xihHANDLEtoSUBPOOLFn
       --> xihGetConnSPDetailsFromList
        --> xihGetConnSPDetails
        <-- xihGetConnSPDetails rc=OK
       <-- xihGetConnSPDetailsFromList rc=OK
      <-- xihHANDLEtoSUBPOOLFn rc=OK
      --> xllSpinLockSlowRequest
      <-- xllSpinLockSlowRequest rc=OK
      --> xllSpinLockRelease
      <-- xllSpinLockRelease rc=OK
      --> xllFreeSem
      <-- xllFreeSem rc=OK
      --> vms_mtx
       --> vms_get_lock
       <-- vms_get_lock rc=OK
      <-- vms_mtx rc=OK
      --> xcsFreeQuickCell
       --> xllSpinLockRequest
       <-- xllSpinLockRequest rc=OK
       --> xstFreeCell
       <-- xstFreeCell rc=OK
       --> xllSpinLockRelease
       <-- xllSpinLockRelease rc=OK
      <-- xcsFreeQuickCell rc=OK
     <-- xllCSCloseMutex rc=OK
    <-- xclDeleteMutexMem rc=OK
    --> xstSerialiseExtent
```

```
                        --> xllSpinLockRequest
                        <-- xllSpinLockRequest rc=OK
                       <-- xstSerialiseExtent rc=OK
                       --> xstFreeChunk
                        --> xstDeleteChunk
                        <-- xstDeleteChunk rc=OK
                        --> xstInsertChunk
                        <-- xstInsertChunk rc=OK
                       <-- xstFreeChunk rc=OK
                       --> xstReleaseSerialisationOnExtent
                        --> xllSpinLockRelease
                        <-- xllSpinLockRelease rc=OK
                       <-- xstReleaseSerialisationOnExtent rc=OK
                      <-- xstFreeBlockInExtent rc=OK
                     <-- xstFreeBlockFromSharedMemSet rc=OK
                    <-- xstFreeMemBlock rc=OK
                   <-- xcsFreeMemBlock rc=OK
                  <-- zxcCleanupAgent rc=OK
                  --> xcsReleaseMutexSem
                   --> xllSemRel
                    --> vms_mtx
                     --> vms_get_lock
                     <-- vms_get_lock rc=OK
                    <-- vms_mtx rc=OK
                   <-- xllSemRel rc=OK
                  <-- xcsReleaseMutexSem rc=OK
                  --> xcsCheckProcess
                   --> kill
                   <-- kill rc=OK
                  <-- xcsCheckProcess rc=OK
                  --> xcsCheckProcess
                   --> kill
                   <-- kill rc=OK
                  <-- xcsCheckProcess rc=OK
                  --> xcsRequestMutexSem
                   --> xllSemReq
                    --> vms_mtx
                     --> vms_get_lock
                     <-- vms_get_lock rc=OK
                    <-- vms_mtx rc=OK
                   <-- xllSemReq rc=OK
                  <-- xcsRequestMutexSem rc=OK
                  --> xcsReleaseMutexSem
                   --> xllSemRel
                    --> vms_mtx
                     --> vms_get_lock
                     <-- vms_get_lock rc=OK
                    <-- vms_mtx rc=OK
                   <-- xllSemRel rc=OK
                  <-- xcsReleaseMutexSem rc=OK
                  --> xcsCheckProcess
                   --> kill
                   <-- kill rc=Unknown(FFFF)
                  <-- xcsCheckProcess rc=xecP_E_INVALID_PID
                  --> xcsBuildDumpPtr
                   --> xcsGetMem
                   <-- xcsGetMem rc=OK
                  <-- xcsBuildDumpPtr rc=OK
                 <-- xcsBuildDumpPtr rc=OK
                <-- xcsBuildDumpPtr rc=Unknown(4B)
                --> xcsFFST

ECAnchor
6A91B0                             5A584541                ZXEA
6A91C0   03000000  E8030000  03220000  0100DA01    ....&#232;...."....&#218;.
6A91D0   2C05A500  D4040000  0A00DA01  01000000    ,.¥.&#212;.....&#218;.....
6A91E0   B0000000  0A00DA01  03000000  E8030000    °.....&#218;.....&#232;...
6A91F0   03220000  0100DA01  07000000  00000000    ."....&#218;.........
6A9200   283F9700  03000000  F0030000  08410000    (?—..........A..
6A9210   0300DA01  03000000  F0030000  08410000    ..&#218;.....¸....A..
6A9220   0300DA01  01000000  B4000000  0200DA01    ..&#218;..... .....&#218;.
6A9230   03000000  E8030000  03220000  0100DA01    ....&#232;...."....&#218;.
6A9240   00000000  00000000  00000000  00000000    ................
6A9250   00000000  00000000  00000000  00000000    ........¸.......
6A9260   00000000  01000000  B4000000  0200DA01    ........ .....&#218;.
6A9270   03000000  E8030000  03220000  0100DA01    ¸...&#232;...."..¸.&#218;.
6A9280   B41F0000  0200DA01  01000000  B4000000     .....&#218;..... ...
6A9290   0200DA01  03000000  E8030000  03220000    ..&#218;.....&#232;....".. 
6A92A0   0100DA01  DA012020  00000000  EFCD0300    ..&#218;.&#218;.  .....&#205;..
6A92B0   00000000  00000000  00000000  00000000    ...............
```

```
6A92C0   00000000  00000000  01000000  00000000    ................
6A92D0   44454641  554C5400  00000000  00000000    DEFAULT.........
6A92E0   00000000  00000000  00000000  00000000    ................
6A92F0   00000000  00000000  00000000  00000000    ................
6A9300   2F6D7173  5F726F6F  742F6D71  6D000000    /mqs_root/mqm...
6A9310   00000000  00000000  00000000  00000000    ................
6A9320 to 6A93F0 suppressed, lines same as above
6A9400   5A435048  01000000  B8000000  0400DA01    ZCPH....&#184;.....&#218;.
6A9410   03000000  F0030000  08410000  0300DA01    .........A....&#218;.
6A9420   DC040000  0400DA01  01000000  B8000000    &#220;.....&#218;.....&#184;...
6A9430   0400DA01  03000000  F0030000  08410000    ..&#218;..........A..
6A9440   0300DA01  00000000  00000000  00000000    ..&#218;.............
6A9450   00000000  00000000  00000000  00000000    ................
6A9460 to 6A9470 suppressed, lines same as above
6A9480   00000000  00000000  5A435048  01000000    ........ZCPH....
6A9490   B8000000  0500DA01  03000000  F0030000    &#184;.....&#218;.........
6A94A0   08410000  0300DA01  DC040000  0500DA01    .A....&#218;.&#220;.....&#218;.
6A94B0   01000000  B8000000  0500DA01  03000000    ....&#184;.....&#218;.....
6A94C0   F0030000  08410000  0300DA01  4C4B0000    .....A....&#218;.LK..
6A94D0   0500DA01  01000000  B8000000  0500DA01    ..&#218;.....&#184;.....&#218;.
6A94E0   03000000  F0030000  08410000  0300DA01    .........A....&#218;.
6A94F0   07000000  00090000  50140000  0100DA01    ........P.....&#218;.
6A9500   01000000  B0000000  0100DA01  03000000    ....°.....&#218;.....
6A9510   E8030000  03220000  0100DA01  98170000    &#232;...."..¸.&#218;.˜...
6A9520   0200DA01  01000000  B4000000  0200DA01    ..&#218;.....´.....&#218;.
6A9530   03000000  E8030000  03220000  0100DA01    ....&#232;...."....&#218;.
6A9540   00000000  00000000  08000000  3C0A0000    ............<...
6A9550   50140000  0100DA01  01000000  B0000000    P.....&#218;.....°...
6A9560   0100DA01  03000000  E8030000  03220000    ..&#218;.....&#232;...."..
6A9570   0100DA01  08180000  0200DA01  01000000    ¸.&#218;.......&#218;.....
6A9580   B4000000  0200DA01  03000000  E8030000    ´.....&#218;.....&#232;...
6A9590   03220000  0100DA01  00000000  00000000    ."....&#218;.........
6A95A0   00000000  00000000  00000000  00000000    ................
6A95B0 to 6A9650 suppressed, lines same as above
6A9660   01000000  00000000  78180000  0200DA01    ....¸...x.....&#218;.
6A9670   01000000  B4000000  0200DA01  03000000    ....´.....&#218;.....
6A9680   E8030000  03220000  0100DA01  09000000    &#232;...."....&#218;.....
6A9690   780B0000  50140000  0100DA01  01000000    x...P.....&#218;.....
6A96A0   B0000000  0100DA01  03000000  E8030000    °.....&#218;.....&#232;...
6A96B0   03220000  0100DA01  00000000  00000000    ."....&#218;.........
6A96C0   00000000  00000000  00000000  00000000    ................
6A96D0 to 6A9750 suppressed, lines same as above
6A9760   00000000  00000000  DD012020  00000000    ........&#221;.  ....
6A9770   00000000  00000000  01000000  09000000    ................
6A9780   00000000                                  ....
```

The Function Stack and Trace History are used by IBM to assist in problem determination. In most cases there is little that the system administrator can do when an FFST is generated, apart from raising problems through the support centers.

However, there is one set of problems that they may be able to solve. If the FFST shows “quota exceeded” or “out of space on device” descriptions when calling one of the internal functions, it is likely that the relevant SYSGEN parameter limit has been exceeded.

To resolve the problem, adjust the system parameters to increase the internal limits. See “Chapter 13. Configuring MQSeries” on page 159 for further details.

Problem determination with clients

An MQI client application receives MQRC_* reason codes in the same way as non-client MQI applications. However, there are now additional reason codes for error conditions associated with clients. For example:

- Remote machine not responding
- Communications line error
- Invalid machine address

Client problem determination

The most common time for errors to occur is when an application issues an MQCONN and receives the response MQRC_Q_MQR_NOT_AVAILABLE. An error message, written to the client log file, explains the cause of the error. Messages may also be logged at the server depending on the nature of the failure.

Terminating clients

Even though a client has terminated it is still possible for the process at the server to be holding its queues open. Normally, this will only be for a short time until the communications layer detects that the partner has gone.

Error messages with clients

When an error occurs with a client system, error messages are put into the error files associated with the server, if possible. If an error cannot be placed there, the client code attempts to place the error message in an error log in the root directory of the client machine.

OS/2, UNIX, and OpenVMS systems clients

Error messages for OS/2, UNIX, and OpenVMS systems clients are placed in the error logs on their respective MQSeries server systems. Typically, these files appear in the `MQS_ROOT:[MQM.ERRORS]` directory on OpenVMS systems and in `/var/mqm/errors` on UNIX systems.

DOS and Windows® clients

The location of the log file AMQERR01.LOG is set by the MQDATA environment variable. The default location, if not overridden by MQDATA, is:

```
C:\
```

Working in the DOS environment involves the environment variable MQDATA.

This is the default library used by the client code to store trace and error information; it also holds the directory name in which the qm.ini file is stored. (needed for NetBIOS setup). If not specified, it defaults to the C drive.

The names of the default files held in this library are:

AMQERR01.LOG
For error messages.

AMQERR01.FDC
For First Failure Data Capture messages.

Chapter 15. Performance tuning

This chapter discusses how you can tune your OpenVMS system to obtain the best performance from MQSeries.

It is not possible for a product like MQSeries to define values for the various tuning OpenVMS parameters that are correct in all circumstances. The most effective values are determined by the workload for MQSeries itself and the OpenVMS system as a whole. Although the parameter settings given in the *MQSeries for Compaq OpenVMS Alpha, Version 5.1 Quick Beginnings* book provide sensible minimum or initial values, these values may need to increase as the workload for the queue managers grows. The process of doing this is called ″tuning″.

Performance tuning of any OpenVMS systems is described in the *OpenVMS Performance Management* book. Use the information found in that book as well as the following points which are relevant to MQSeries specifically:

- Some tuning parameters apply to the system as a whole (for example, GBLPAGES). These parameters are controlled by the SYSGEN utility and are therefore sometimes called SYSGEN parameters. If used regularly, the AUTOGEN FEEDBACK mechanism monitors the resources being used by the system and adjusts SYSGEN parameters automatically to track the changing workload. This can greatly reduce the manual intervention required to keep the system properly tuned and help avoid errors arising from resource exhaustion. The SYSGEN parameters that are relevant to MQSeries are:

  GBLPAGES, GBLSECTIONS and GBLPAGFIL
  The queue manager is implemented as a set of cooperating processes that communicate via shared (global) memory. Therefore, it is important to ensure that the SYSGEN parameters that control global memory (GBLPAGES, GBLSECTIONS and GBLPAGFIL) are sufficiently large. The *MQSeries for Compaq OpenVMS Alpha, Version 5.1 Quick Beginnings* book provides sensible initial values for these parameters. As the number of users of MQSeries grows, the demand for global memory increases and you may need to increase the corresponding SYSGEN parameters as well.

  CHANNELCNT
  The queue manager processes use the OpenVMS mailboxes as an interprocess communication and synchronization mechanism. These mailboxes are accessed via channels and so it is necessary to ensure that the SYSGEN parameter CHANNELCNT is large enough. In most cases the value set during the installation will be sufficient. However, in heavily-loaded systems with many active MQSeries processes, the value may have to been increased.

  To set SYSGEN parameters explicitly, modify the file MODPARAMS.DAT as described in the *OpenVMS Performance Management* book.

- Some OpenVMS tuning parameters apply to individual user names or processes (for example, PGFLQUOTA). These parameters are typically (but not always) controlled by the AUTHORIZE utility. There is no automatic method for adjusting these parameters. However, since these parameters represent a limit on the amount of some resource that can be used by a particular process, you may want to set them higher than strictly necessary to provide spare capacity for occasional peak loads. The process specific parameters that are relevant to MQSeries are:

PGFLQUOTA
: This controls the amount of pagefile space that a process is allowed to use. Since MQSeries processes are typically moving messages that may be very large or numerous or both, they could potentially consume large amounts of pagefile space.

PRCLM
: This parameter controls the number of subprocesses that a given process can create. Since most MQSeries processes are created as subprocesses of the execution controller, the system will require a high value for PRCLM.

ENQLM, ASTLM, TQELM
: As mentioned previously, the queue manager is implemented as a set of cooperating processes. These processes synchronize their activities using the OpenVMS lock manager, asynchronous system traps (ASTs) and timers. The three parameters that limit use of these resources must be set to a large enough value to cope with the needs of the queue manager.

Setting the value of process specific parameters

The most common way to set these parameters is to use the AUTHORIZE utility to adjust the value for the appropriate user name (which for MQSeries is usually MQM).

However, on OpenVMS some process quotas are shared by all processes in the same job, that is, a parent process and all other processes that are subprocesses of the parent. Process quotas in this category include BYTLM, FILLM, PGFLQUOTA, PRCLM, TQELM and ENQLM and these are called *pooled quotas*.

Since most queue manager processes are created as subprocesses of the execution controller, it follows that the pooled quotas are shared by all queue manager processes. Therefore, it is usually necessary to set these quotas to values that seem excessive for a single process. This is particularly true of PGFLQUOTA since this parameter limits the amount of virtual memory that the queue manager processes can **collectively** create. For this reason, when the execution controller starts, it does not obtain its initial quota values from the authorization file maintained by AUTHORIZE, but sets them explicitly itself to reasonable values. As a result of this, AUTHORIZE can no longer be used to modify these values. Instead you can override the explicit quota values using the following logical names:

```
MQS_ASTLM
MQS_BIOLM
MQS_BYTLM
MQS_DIOLM
MQS_ENQLM
MQS_FILLM
MQS_PGFLQUOTA
MQS_PRCLM
MQS_TQELM
```

Setting process specific parameters

You can set these logicals using a file called `SYS$MANAGER:MQS_SYSTARTUP.COM`. MQSeries provides a file called `SYS$MANAGER:MQS_SYSTARTUP.TEMPLATE` that can be edited and renamed. For example, to provide a different value for the PGFLQUOTA parameter:

1. Copy the `.TEMPLATE` file, `MQS_SYSTARTUP.TEMPLATE` to a `.COM` file.
2. Edit the `MQS_SYSTARTUP.COM` file you have just created to uncomment (activate) the lines that define the logicals corresponding to the process quotas such as mqs_pgflquota.
3. Define new values.

   For example:

```
$! DEFINE/SYSTEM MQS_PGFLQUOTA 1000000
```

   might become

```
$  DEFINE/SYSTEM MQS_PGFLQUOTA 5000000
```

4. Invoke the `mqs_systartup` file to define the logicals. For example:

```
$ @sys$manager:mqs_systartup
```

   Typically, this is done as part of the system startup procedure.

Shortages of pooled quotas typically become apparent when either a new client application fails to connect to the queue manager or some other application fails because the new connection has consumed too much resource.

Also note that since the execution controller is started with explicit settings for many of its quotas, the SYSGEN PQL_D* parameters do not apply to the EC.

Chapter 16. MQSeries for OpenVMS and clustering

OpenVMS clusters and *MQSeries queue manager clusters* are two different things, independent of one another.

Note: When the term *cluster* is used, it refers to an MQSeries queue manager cluster. An OpenVMS cluster is always referred to as *OpenVMS cluster*.

MQSeries queue manager clusters do not necessarily use OpenVMS cluster intercommunication protocols, the OpenVMS cluster distributed lock manager or the OpenVMS cluster file system. All communication between queue managers in an MQSeries cluster is via MQSeries channels using one of the supported protocols. Thus it is possible to configure MQSeries queue manager clusters with queue managers that run on OpenVMS systems that are not part of the same OpenVMS cluster.

If an MQSeries queue manager is configured within an OpenVMS cluster, the MQSeries queue manager can run only on one OpenVMS node (referred to as a node for the rest of this chapter) within the OpenVMS cluster at a time. The function of a single MQSeries queue manager cannot be distributed across multiple OpenVMS nodes within an OpenVMS cluster. If an attempt is made to start an MQSeries queue manager on more than one OpenVMS node, an error is returned. However, if there are multiple MQSeries queue managers configured in an OpenVMS cluster they can be run on different OpenVMS nodes within the OpenVMS cluster.

To provide a higher level availability of MQSeries queue managers in an OpenVMS cluster a new feature called Failover Sets has been introduced in MQSeries V5.1. This enables a queue manager to be automatically restarted on another OpenVMS cluster node if a failure occurs. This feature can be used with or without MQSeries queue manager clusters. (See "OpenVMS cluster failover sets" on page 204).

Installing MQSeries in an OpenVMS cluster

Installing MQSeries for Compaq OpenVMS Alpha, V5.1 in an OpenVMS cluster is very similar to installing MQSeries on a standalone OpenVMS system. However, before installing, you need to consider the following:

- If there are multiple system disks in the OpenVMS cluster, MQSeries needs to be installed on each system disk that has a node booted from it *and* that has to run MQSeries. MQSeries needs to be installed only once per system disk, not once per node.
- The disk holding the MQS_ROOT directory structure must be mounted system wide on the OpenVMS nodes that are to run the queue managers contained within the directory structure. It is possible to have different MQS_ROOT directory structures for each node. But, if failover sets are to be configured, each OpenVMS node in a failover set must refer to the same MQS_ROOT directory structure. When installing MQSeries, you must specify the MQS_ROOT directory (in response to the question 'Enter the root device for the MQSeries datafiles:') for each installation.

Installing in an OpenVMS cluster

- If the disk containing the log files for a queue manager is different from the disk containing MQS_ROOT, the disk containing the log files must be mounted system-wide on all nodes in a Failover set.
- MQSeries uses an account MQM which has a default directory SYS$SPECIFIC:[MQS_SERVER]. This directory is created only for the node on which MQSeries is installed. The directory must be created for each additional node that boots from the same system disk *and* that has to run MQSeries. This can be done by executing the following DCL commands on each additional node:

```
$create/directory sys$specific:[mqs_server]/owner=[mqs_server] -
/protection=(s:rwed,o:rwed,g,w)
$set sec/acl=(identifier=mqm,options=default,access=r+w+e+d+c) -
sys$specific:[000000]mqs_server.dir
$set sec/acl=(identifier=mqm,access=r+w+e+d+c) -
sys$specific:[000000]mqs_server.dir
```

OpenVMS cluster failover sets

Overview of OpenVMS cluster failover sets

OpenVMS cluster failover sets are a new feature available in MQSeries for Compaq OpenVMS, V5.1. They allow MQSeries queue managers to be automatically restarted on another OpenVMS node in an OpenVMS cluster if the MQSeries queue manager fails. The following types of failure are supported by OpenVMS cluster failover sets:

- Halt of an OpenVMS node running an MQSeries queue manager
- System crash of an OpenVMS node running an MQSeries queue manager
- Shutdown of an OpenVMS node running an MQSeries queue manager without the clean ending of the MQSeries queue manager
- The failure of an MQSeries queue manager Execution Controller process

The following types of failure are not supported by OpenVMS cluster failover sets:

- A fault on an OpenVMS node running an MQSeries queue manager that does not cause the node or the MQSeries queue manager to fail.
- The failure of an MQSeries queue manager process except for the Execution Controller process. An MQSeries queue manager is never automatically restarted on the same node.
- A software or hardware failure of the disk holding the MQSeries queue manager queue files and log data.
- Corruption of the MQSeries queue manager queue files or log data.

OpenVMS cluster failover sets are supported only for queue managers that use the TCP/IP protocol for MQSeries channels. The following TCP/IP stacks are supported:

- Digital® TCP/IP Services for OpenVMS V5.0A
- Porcess Software's TCPware® for OpenVMS V5.4
- Process Software's Multinet® for OpenVMS 4.3

OpenVMS cluster failover set concepts

An OpenVMS cluster failover set is a collection of OpenVMS nodes that can potentially run an MQSeries queue manager. There may be between one and four

OpenVMS cluster failover sets concepts

OpenVMS nodes in an OpenVMS cluster failover set and all the OpenVMS nodes must be members of the same OpenVMS cluster. An OpenVMS cluster failover set is specific to one MQSeries queue manager. There may be more than one OpenVMS cluster failover set configured in an OpenVMS cluster. Note that the maximum length of a queue manager name supported by OpenVMS cluster failover sets is 25 characters.

Failover is the process by which an MQSeries queue manager is restarted on another OpenVMS node when a supported failure occurs. After this process has completed the MQSeries queue manager is said to have failed over.

Failback is the process by which an MQSeries queue manager is restarted on its original OpenVMS node after a failure has been resolved. OpenVMS cluster failover sets do not support automatic failback but it can be performed manually. After this process has completed the MQSeries queue manager is said to have failed back.

A *failover monitor* is a process that runs on each member of an OpenVMS cluster failover set. The failover monitors are responsible for performing all functions of the failover sets. The failover monitors within an OpenVMS cluster failover set cooperate with one another to provide these functions. A failover monitor is started using the **runmqfm** command. (For more on this command, see "runmqfm (Start a failover monitor)" on page 273.)

One failover monitor is nominated as the *watcher failover monitor* and this failover monitor is said to be in a *watching* state. The first failover monitor to start in a failover set is the initial watcher failover monitor. A failover set becomes *live* when the first failover monitor is started. If the watcher failover monitor fails, or the OpenVMS node on which it is running fails, another failover monitor is automatically nominated as the watcher failover monitor. The watcher failover monitor is responsible for checking that the MQSeries queue manager is running and for initiating a failover operation if a supported failure occurs. Any operation that must be performed on another OpenVMS node is forwarded by the watcher failover monitor to the failover monitor running on the relevant OpenVMS node which actually performs the operation.

OpenVMS cluster failover sets are administered using the DCL command **failover**. The **failover** command can be used from any node in the OpenVMS cluster failover set. All commands are sent to the watcher failover monitor which then decides which failover monitor should process the command and if necessary forwards it onto another failover monitor.

The OpenVMS cluster failover set configuration file holds the details of the OpenVMS cluster failover set including the number and names of the OpenVMS nodes. The file is called `FAILOVER.INI` and resides in the directory `MQS_ROOT:[MQM.QMGRS.queuemanagername]`. It is a text file which is modified with a text editor and must be created prior to starting the first failover monitor. A template configuration file called `FAILOVER.TEMPLATE` is provided in the directory `MQS_EXAMPLES`. The parameters in the configuration file cannot be changed dynamically. For a change to take effect all failover monitors must be stopped and then started again. Care must be taken when this is done because automatic failover of the MQSeries queue manager cannot occur when the failover monitors are not started.

For an MQSeries queue manager in a failover set, all MQSeries commands continue to work as normal except for the **strmqm** and **endmqm** commands. These two commands return an error when an MQSeries queue manager is in a live failover set. The **failover** command must be used to start and end the MQSeries queue manager.

The OpenVMS node priority is the priority given to each OpenVMS node in the OpenVMS cluster failover set and is used to determine on which OpenVMS node the queue manager should be started after a failure has occurred. The OpenVMS node with the lowest numeric priority value has the highest priority.

The OpenVMS cluster failover set TCP/IP address is the TCP/IP address assigned to the failover set. All channels that refer to the failover set queue manager must be configured to specify this TCP/IP address in the connection name. Each OpenVMS cluster failover set must use a unique TCP/IP address. All OpenVMS nodes in the OpenVMS cluster failover set must have an interface TCP/IP address in the same subnet and the OpenVMS cluster failover set TCP/IP address must be in the same subnet.

Preparing to configure an OpenVMS cluster failover set

The following steps must be taken before configuring an OpenVMS cluster failover set:

1. Create the queue manager using **crtmqm** if it does not already exist.
2. Obtain a TCP/IP address for the OpenVMS cluster failover set.
3. Create or modify MQSeries channels to use the OpenVMS cluster TCP/IP failover set TCP/IP address.
4. Decide on the OpenVMS nodes that are to be in the OpenVMS cluster failover set and decide their priorities.
5. Ensure that the MQS_ROOT logical refers to the same directory on all OpenVMS nodes in the OpenVMS cluster failover set and that the disk is mounted system-wide on all of the nodes. The disks containing the `MQS_ROOT` directory and the log files should not be MSCP served from one node in the failover set to another node because if the node serving the disk becomes unavailable, the node to which the disk is served will no longer be able to access the disks.

Configuring an OpenVMS cluster failover set

The following steps are required to configure an OpenVMS cluster failover set:

1. Copy the `MQS_EXAMPLES:FAILOVER.TEMPLATE` file to `MQS_ROOT:[MQM.QMGRS.`*queuemanagername*`]FAILOVER.INI.`
2. Edit the `MQS_ROOT:[MQM.QMGRS.`*queuemanagername*`]FAILOVER.INI` file and modify for this OpenVMS cluster failover set configuration. (See “Editing the FAILOVER.INI configuration file” on page 207.)
3. Edit the **START_QM.COM, END_QM.COM** and **TIDY_QM.COM** command procedures. (See “Command procedures used by failover sets” on page 208.)
4. Set up ICC security for the ICC Association used by the failover monitors (See “Setting up security for ICC associations” on page 214.)
5. Start a failover monitor on each node in the OpenVMS cluster failover set using the **runmqfm -m** *queuemanagername* command.
6. Start the queue manager using the **failover -m** *queuemanagername* **-n** *nodename* **-s** command.

7. Modify the site specific shutdown to:
   - End or Move the queue manager if it is running on a node when it is shut down.
   - Halt the failover monitor.

OpenVMS cluster failover set post-configuration tasks

The following are tasks you can perform after your cluster failover set has been configured:

- Edit the system startup file to start the failover monitor processes on each node in the OpenVMS cluster failover set using the **runmqfm** command. The **runmqfm** command should be placed after the command to start MQSeries.
- If it is required to start the queue manager automatically on system startup, place a command in the system startup on the relevant node to start the queue manager after the failover monitor has been started. The command to start the queue manager on a node is **failover -m** *queuemanagername* **-n** *nodename* **-s**.
- Modify the site specific shutdown to end the failover monitor on shutdown of the system. Also End or Move the queue manager if it is running on a node when it is shut down.

Editing the FAILOVER.INI configuration file

The `FAILOVER.INI` file must be customized for each OpenVMS cluster failover set. The meaning of each of the fields is listed in Table 9 on page 207. The template configuration file supplied in `MQS_EXAMPLES` is included the "Appendix F. OpenVMS cluster failover set templates" on page 311. Any line in the file that begins with a '#' character is ignored when it is read by a failover monitor process. The character case of the field names within the file must be as specified in the template file. Each field name must be followed by an '=' character and then the associated value. All the fields in the template file are mandatory so no fields must be removed.

Table 9. Description of the fields within the FAILOVER.INI file

Field name	Description
IpAddress	The TCP/IP address to be used by the failover set
PortNumber	The TCP/IP port number used by the listener for the queue manager
TimeOut	This time out value is passed to the EndCommand procedure. See "Command procedures used by failover sets" on page 208.
StartCommand	The command procedure used to start the queue manager
EndCommand	The command procedure used to end the queue manager
TidyCommand	The command procedure used to tidy up on a node after a queue manager failure in which the OpenVMS node survives
LogDirectory	The directory which holds the log files created by the StartCommand, EndCommand and TidyCommand procedures
NodeCount	The number of nodes in the failover set. The number of node triplets defined after this field must correspond to this value. The maximum number of nodes supported is four.
NodeName	The node name of the node. That is the value specified for the SCSNODE OpenVMS system parameter.

Table 9. Description of the fields within the FAILOVER.INI file (continued)

Field name	Description
Interface	The TCP/IP interface name for the node when using the Digital TCP/IP Services for OpenVMS TCP/IP stack. This can be obtained from the output of the $tcpip show interface command. This field is not used when using the TCPware for OpenVMS TCP/IP or Multinet for OpenVMS TCP/IP stacks but the default value of we0 should still be specified. (Do not remove the field from the configuration file.)
Priority	This is the priority given to this node within the failover set. The value must be between 1 and 10. A value of 1 is the highest priority. Multiple nodes can have the same priority. When a failure occurs or no specific node is specified in a **failover** -s or -f command, the queue manager is started on the highest priority node that is available.

Command procedures used by failover sets

Failover sets use three command procedures to implement some of its functions. The locations of these command procedures are specified in the `FAILOVER.INI` configuration file by the field names StartCommand, EndCommand and TidyCommand. Template files for these command procedures, with names **START_QM.TEMPLATE**, **END_QM.TEMPLATE** and **TIDY_QM.TEMPLATE** respectively, are provided in `MQS_EXAMPLES`. These files are listed in “Appendix F. OpenVMS cluster failover set templates” on page 311.

The command procedures are passed five or six parameters. These are listed in Table 10 on page 208:

Table 10. Parameters passed to command procedures

Parameter	Value
P1	Queue manager name
P2	Queue manager directory name
P3	Cluster TCP/IP address
P4	Node Interface name
P5	Listener port number
P6	End queue manager timeout (EndCommand procedure only)

The StartCommand procedure is used to start the queue manager in the following circumstances:

- When explicitly specified with the -s flag of the **failover** command
- When the queue manager is moved to another OpenVMS node using the -f flag of the **failover** command.
- When automatically restarted after a queue manager failure

By default the StartCommand procedure configures the failover set TCP/IP address on the node to run the Queue manager and then starts the queue manager using the **strmqm -m** *queuemanagername* command. Depending on system requirements the command procedure can be modified in the following ways:

- Change the **strmqm** command
- Add commands to start additional MQSeries processes such as the listener

- Add commands to start application processes

The StartCommand procedure must exit with a status of 1 for the queue manager to be monitored after the queue manager has started.

The EndCommand procedure is used to end the queue manager in the following circumstances:

- When explicitly specified with the -e flag of the **failover** command
- When the queue manager is moved to another OpenVMS node using the -f flag of the **failover** command

By default the EndCommand procedure attempts to end the queue manager with the **endmqm -i** *queuemanagername* command. If the queue manager has not ended within the timeout period specified in the configuration file, the procedure attempts to end the queue manager with the **endmqm -p** *queuemanagername* command. If the queue has still not ended within another timeout period, the queue manager is ended by deleting the Execution Controller process. Once the Queue manager is ended, the failover set TCP/IP address is deconfigured. If the queue manager is successfully ended using the **endmqm** commands, the status SS$_NORMAL is returned. If the queue manager is ended by deleting the Execution Controller, the status SS$_ABORT is returned. If the queue manager is not ended after a third timeout period, the status SS$_TIMEOUT is returned. These statuses are used by the watcher failover monitor to determine the outcome of the EndCommand procedure and sets the state of the failover set accordingly. Depending on system requirements the command procedure can be modified in the following ways:

- Add commands to end additional MQSeries processes such as the listener
- Add commands to end Application processes

The TidyCommand procedure is used to tidy up on an OpenVMS node if the queue manager fails but the OpenVMS node continues to run.

By default the TidyCommand procedure deconfigures the failover set TCP/IP address. Depending on system requirements the command procedure can be modified in the following ways:

- Add commands to end any MQSeries processes that are still running such as the listener
- Add commands to end Application processes that are still running

The template files are set up by default to use Digital TCP/IP Services for OpenVMS commands to configure and de-configure the TCP/IP address. If you are using TCPware for OpenVMS or MultiNet for OpenVMS, comment out (deactivate) the Digital TCP/IP Services for OpenVMS commands and uncomment (activate) the TCPware for OpenVMS or MultiNet for OpenVMS commands.

Administration of failover sets

Failover sets must be managed from the SYSTEM account or from an MQSeries Administration account. Failover sets are managed using two commands DCL **runmqfm** and **failover**. The **runmqfm** command is used to start the **failover** monitors and the **failover** command performs all other administration tasks. These commands are described in “runmqfm (Start a failover monitor)” on page 273 and “failover (Manage a failover set)” on page 258.

Startup of failover monitors

Failover monitors are started by executing the **runmqfm** command on the OpenVMS node on which it is required to have the failover monitor started. For example to start a failover monitor for queue manager TESTQM, use the command:

```
 $ runmqfm -m TESTQM
```

This creates a detached process with a name based on the queue manager name and ending in _FM. In this example the process name is TESTQM_FM. This process is listed in a monmq active display.

If a log file is required, this can be specified by redirecting output of the **runmqfm** command and additional debug information can be displayed in the log file by specifying the -d flag. For example:

```
$ runmqm -m TESTQM -d > sys$manager:fm.log
```

Note that the **runmqfm** command only starts failover monitor processes it does not start the queue manager.

Starting a queue manager within a failover set

To start a queue manager within a failover set, at least one failover monitor must be running and there must be a failover monitor running on the node on which you wish to start the queue manager. A queue manager is started using the -s flag of the **failover** command. The command can be executed from any OpenVMS node within the failover set. For example if you wish to start the queue manager TESTQM on node BATMAN, use the following command:

```
$ failover -m TESTQM -n BATMAN -s
```

If it is required to start the queue manager on the OpenVMS node with the highest priority available omit the -n flag from the command. For example:

```
$ failover -m TESTQM -s
```

Note that once a failover monitor is started for a queue manager (on any node), any attempt to use the **strmqm** command to start a queue manager will fail. However, once all failover monitors have been stopped for a queue manager, the **strmqm** command can be used normally.

Ending a queue manager within a failover set

To end a queue manager within a failover set, there must be a failover monitor running on the node on which the queue manager is running. A queue manager is ended using the -e flag of the **failover** command. The command can be executed from any OpenVMS node within the failover set. For example if you wish to end the queue manager TESTQM, use the following command:

```
$ failover -m TESTQM -e
```

Note that once a failover monitor is started for a queue manager (on any node), any attempt to use the **endmqm** command to end a queue manager will fail. However, once all failover monitors have been stopped for a queue manager, the **endmqm** command can be used normally.

Moving a queue manager within a failover set

Moving a queue manager within a failover set means stopping the queue manager on the node on which it is currently running and then starting it again on another node within the failover set. To move a queue manager within a failover set, there must be a failover monitor running on the node on which the queue manager is currently running and there must be a failover monitor running on the node on which you wish to move the queue manager.

A queue manager is moved using the -f flag of the **failover** command. The command can be executed from any OpenVMS node within the failover set. For example if you wish to move the queue manager `TESTQM` to node ROBIN, use the following command:

```
$ failover -m TESTQM -n ROBIN -f
```

If it is required to move the queue manager to the OpenVMS node with the highest priority available, omit the -n flag from the command. For example:

```
 $ failover -m TESTQM -f
```

Displaying the state of a failover set

There are three types of state that the describe the over all state of the failover set:
- The failover set queue manager state
- The failover set node queue manager states (one for each node)
- The failover set Node Monitor states (one for each node)

The possible values of each of the states are described in the following three tables.

Table 11. Failover set queue manager states

State	Description
STOPPED	The queue manager has never been started in the failover set or has been cleanly shutdown
STARTED	The queue manager has been started in the failover set. The failover set will try to restart the queue manager if a queue manager failure occurs.

Table 12. Failover set node queue manager states

State	Description
AVAILABLE	The node is free to have the queue manager started on it if a failure occurs on another node.
RUNNING	The queue manager is running on this node.

Table 12. Failover set node queue manager states (continued)

State	Description
EXCLUDED	The queue manager was stopped on this node in an unclean manner without the node itself failing. If a queue manager fails on another node it will not be restarted on this node.

Table 13. Failover set node monitor states

State	Description
STARTED	A failover monitor is running on this node but it is not the watcher.
WATCHING	A failover monitor is running on this node and it is the watcher.
STOPPED	There is no failover monitor running on this node.

The state of a failover set is displayed using the -q flag of the **failover** command. There must be at least one failover monitor process running and the command can be executed from any node within the failover set. For example to display the state of the failover set for the queue manager TESTQM, use the following command:

```
$ failover -m TESTQM -q
```

Sample output from the command is shown below:

```
83H8439, 5697-270 (C) Copyright IBM Corp. 1996.  ALL RIGHTS RESERVED.

OpenVMS Cluster Failover Set - Configuration and State.

Queue Manager Name                 : TESTQM
Sequence No                        : 11
TCP/IP Address                     : 10.20.30.40
Listener Port Number               : 1414
Timeout to end the Queue Manager   : 30
Queue Manager state in Failover Set : STARTED

OpenVMS Node - Configuration and State

Node name              : BATMAN
Priority               : 2
TCP/IP Interface       : we0
Queue Manager state    : RUNNING
Failover Monitor state : WATCHING

Node name              : ROBIN
Priority               : 1
TCP/IP Interface       : we0
Queue Manager state    : EXCLUDED
Failover Monitor state : STARTED
```

Setting DCL symbols to the state of a failover set

In some cases it may be necessary to write DCL command procedures to control failover sets. The -l flag of the **failover** command sets three local DCL symbols to indicate the state of the failover set. These symbols can then be used to take conditional actions based on the state of the queue manager. There must be at least one failover monitor process running and the command can be executed from any node within the failover set. The symbols that are set are shown in Table 14 on page 213 page 213.

Table 14. DCL symbols and description

DCL symbol name	Description
MQS$QMGR_NODE	Set to the OpenVMS node that is running the queue manager or a null string if there is no queue manager running
MQS$AVAILABLE_NODES	Set to the list of OpenVMS nodes that are available to run the queue manager. That is the nodes that are in the queue manager AVAILABLE state and that have a failover monitor running.
MQS$MONITOR_NODES	Set to the list of OpenVMS nodes that have a failover monitor running on them.

For example to set the symbols to the state of the failover set for the queue manager `TESTQM`, use the following command:

```
$ failover -m TESTQM -l
```

Example results for the setting of the symbols are shown below:

```
MQS$AVAILABLE_NODES = ""
MQS$MONITOR_NODES = "BATMAN,ROBIN"
MQS$QMGR_NODE = "BATMAN"
```

Halting a failover monitor process

The failover monitor process on an OpenVMS node can be halted using the -h flag of the **failover** command. The command can be executed from any node within the failover set. For example to halt the failover monitor for queue manager `TESTQM` on node BATMAN use the following command:

```
$ failover -m TESTQM -n BATMAN -h
```

If the failover monitor being halted is the watcher failover monitor, another failover monitor becomes the watcher if one exists. If the failover monitor being halted is the last failover monitor for the failover set, the failover set will no longer be live. In this case, the queue manager can now be started and ended using the **strmqm** and **endmqm** commands. The -h flag of the **failover** command never ends a queue manager. If the queue manager is running on the OpenVMS node on which the failover monitor is being halted, the queue manager will continue to run.

Executing commands while an update is in progress

The **failover** commands with flags -s, -e, -f and -c are considered updates. While these commands are in progress, an update in progress flag is set by the watcher failover monitor. When this flag is set, any other update and failover monitor halt command will fail because simultaneous updates are not allowed. Non-update commands such as the -q and -l flags continue to work when an update is in progress.

In rare circumstances, a failed update may leave the update in progress flag set. The -u flag of the **failover** command clears the update in progress flag. This command should be used with caution. For example to clear the update in progress flag for queue manager `TESTQM`, use the following command:

```
$ failover -m TESTQM -u
```

Changing the state of a failover set

In some circumstances it may be necessary to change the state of a failover set. This is achieved using the -c flag of the **failover** command. This is most likely needed when a queue manager state on a node is EXCLUDED after a failure and you want to change the state back to AVAILABLE after cleaning up the node. For example, to change the state to AVAILABLE for queue manager `TESTQM` on node BATMAN, use the following command:

```
$ failover -m TESTQM -n BATMAN -c -qmgr available
```

Also you may want to temporarily exclude a node from being considered as a candidate for running the queue manager by changing the Node queue manager state from AVAILABLE to EXCLUDED. For example to change the state to EXCLUDED for queue manager `TESTQM` on node BATMAN, use the following command:

```
$ failover -m TESTQM - n BATMAN -c -qmgr excluded
```

It is also possible to change all of the other states but any change takes effect only if the change requested is consistent with the running system. For instance, if a failover monitor is running on a node and you try to change the Monitor state to STOPPED, this change will not take effect. Apart from changing the Node queue manager states between EXCLUDED and AVAILABLE, it should not be necessary to use the change state command because every 30 seconds the watcher failover monitor performs an integrity check and makes any changes to the states if there is a discrepancy with the running system.

Setting up security for ICC associations

The failover set monitor and client programs use OpenVMS Intra Cluster Communication (ICC) calls to pass messages. To prevent unauthorized users sending messages to the failover monitor processes, the security for the ICC Associations should be configured in the **SYS$STARTUP:ICC$SYSTARTUP.COM** command procedure.

Each failover set uses two association names: one with the name of the queue manager which is used for communication with the watcher failover monitor and the other with the name of the queue manager with `_MQ_FM` appended which is used to communicate with each failover monitor.

An example is shown in Figure 22 on page 215 of the entries required in **ICC$SYSTARTUP.COM** for each node in a failover set. There are two nodes in the failover set called BATMAN and ROBIN and the queue manager name is `TESTQM`.

```
$! ---------------------  List Nodes with Special Actions -------------------
$!
$ nodeactions = "/BATMAN/ROBIN/"
$ if f$locate("/"+nodename+"/",nodeactions) .eq. f$length(nodeactions) -
then goto exit ! No action for this node
$ goto 'nodename' ! Go to action code for this node
$!
$! --------------------  Major Nodes ----------------------
$BATMAN:
$ROBIN:
$!
$! Place in here calls to @SYS$MANAGER:ICC$CREATE_SECURITY_OBJECT and
$! @SYS$MANAGER:ICC$ADD_REGISTRY_TABLE that apply to FAilover odes in the
$! cluster
$!
$!
$  @SYS$MANAGER:ICC$CREATE_SECURITY_OBJECT ICC$::"TESTQM" -
"/owner=MQM/acl=((id=MQM,access=open+access),(id=*,access=none))"
$!
$  @SYS$MANAGER:ICC$CREATE_SECURITY_OBJECT 'nodename'::"TESTQM" -
"/owner=MQM/acl=((id=MQM,access=open+access),(id=*,access=none))"
$!
$  @SYS$MANAGER:ICC$CREATE_SECURITY_OBJECT 'nodename'::"TESTQM_MQ_FM" -
"/owner=MQM/acl=((id=MQM,access=open+access),(id=*,access=none))"
$!
$  set security/class=logical_name_table icc$registry_table -
       /acl=(id=MQM,access=read+write)
$!
$  GOTO EXIT
$!
```

Figure 22. Sample entry required for ICC$SYSTARTUP.COM

Note that ICC Association names are limited to 31 characters, therefore the maximum supported length of MQSeries queue manager name is 25 characters when used in a failover set. Further information on setting up the security of ICC Associations can be found in the *OpenVMS System Manager's Manual*.

Troubleshooting problems with failover sets

When the **start_qm.com**, **end_qm.com** and **tidy_qm.com** procedures are executed, a log file is written to the LogDirectory specified in the `failover.ini` configuration file. The names of the log files are *qmgrname_procedurename*`.log`. For example for a queue manager name of `TESTQM` the **start_qm.com** command procedure will produce a log file with name `testqm_start_qm.log`.

By default the failover monitor does not produce a log file, but a log file can be specified using the redirection parameter on the **runmqfm** command. Additional debug information can be written to the file by specifying the -d parameter on the **runmqfm** command.

Check whether any FDC files have been generated in `MQS_ROOT:[MQM.ERRORS]`.

Using MultiNet for OpenVMS with failover sets

To use Multinet for OpenVMS with failover sets, the cluster alias service must be enabled. The cluster alias service is enabled with command:

```
$ MULTINET CONFIGURE/SERVERS
SERVER-CONFIG> ENABLE CLUSTERALIAS
SERVER-CONFIG> EXIT
```

The template command files assume that there is only one cluster alias address, and that is used by the failover set. However, if there are other cluster alias addresses being used, the command procedures will need to be modified so that the other addresses remain in the MULTINET_CLUSTER_IP_ALIASES logical name.

An example of using failover sets

The following is an example of configuring two nodes, BATMAN and ROBIN, in an OpenVMS cluster into a failover set for queue manager `TESTQM`. The failover set TCP/IP address is 10.20.30.40 and the TCP/IP listener port is 1414. The node BATMAN is nominated as the primary node and ROBIN the secondary node. Initially the queue manager will be started on BATMAN and if the queue manager fails, it will be restarted on ROBIN. If the queue manager is running on ROBIN the queue manager will not be failed back onto BATMAN when ROBIN is rebooted. If the node running the queue manager is shutdown, the queue manager is ended and the failover monitor halted. If the node is not running, the queue manager is not shut down and only the failover monitor is halted.

Customizing failover.template

The failover.template file is modified as follows and copied as `mqs_root:[mqm.qmgrs.testqm]failover.ini`.

```
# FAILOVER.TEMPLATE
# Template for creating a FAILOVER.INI configuration file
# All lines beginning with a '#' are treated as comments
#
# OpenVMS Cluster Failover Set Configuration information
# --------------------------------------------------------
#
# The TCP/IP address used by the OpenVMS Cluster Failover Set
#
IpAddress=10.20.30.40
#
# The TCP/IP port number used by the MQSeries Queue Manager
#
PortNumber=1414
#
# The timeout used by the EndCommand command procedure
#
TimeOut=30
#
# The command procedure used to start the Queue Manager
#
StartCommand=@sys$manager:start_qm
#
# The command procedure used to end the Queue Manager
#
EndCommand=@sys$manager:end_qm
#
# The command procedure used to tidy up on a node after a
# Queue Manager failure but the OpenVMS node did not fail
#
TidyCommand=@sys$manager:tidy_qm
#
# The directory in which the log files for the start, end and
# tidy commands are written
#
LogDirectory=mqs_root:[mqm.errors]
#
# The number of nodes in the OpenVMS Cluster Failover Set. The
# number of nodes defined below must agree with this number
#
NodeCount=2
#
# The Name of the OpenVMS node
#
NodeName=BATMAN
#
# The TCP/IP interface name for the node
#Interface=we0
#
# The priority of the node
#
Priority=1
#
# The Name of the OpenVMS node
#
NodeName=ROBIN
#
# The TCP/IP interface name for the node
#
Interface=we0
#
# The priority of the node
#
Priority=2
```

Figure 23. Failover.template for creating a FAILOVER.INI configuration file

Modification of failover set command procedures

The command procedures are copied from the template files. The only modifications are that the start of the listener in **start_qm.com** and the end of the listener in **end_qm.com** are uncommented (activated). If there are applications to be stopped and started, the appropriate commands could be added to the command procedures.

Example failover set start command procedure, start_failover_set.com

The **start_failover_set.com** command procedure is used to start the failover monitor on each node and conditionally start the queue manager. The procedure is called from the system startup after the **MQS_STARTUP.COM** command procedure has been executed. The procedure is passed two parameters: the queue manager name and the primary node name. In this case it is called as follows:

```
$@start_failover_set testqm batman
```

The **start_failover_set.com** command procedure starts the failover monitor and then uses then -l parameter on the **failover** command to find out the state of the failover set. Note that the failover monitor may not have completely started when the **failover** command is executed so the command is retried up to three times with a second between each attempt. Then if the node is the primary node and the queue manager is not started, it is started using the -s parameter of the **failover** command.

```
$on error then exit
$@sy$manager:mqs_symbols
$!
$! start_failover_set.com
$! ----------------------
$! Command procedure to start a Failover Set Queue Manager during startup
$!
$! p1 = Queue Manager name
$! p2 = Primary Node name
$!
$! Check that the Queue Manager has been specified
$!
$if p1 .eqs ""
$then
$ Write sys$output "Queue Manager name omitted"
$ exit
$else
$ qmgr_name = p1
$endif
$!
$! Check that the primary node name has been specified
$!
$if p2 .eqs ""
$then
$ Write sys$output "Node name omitted"
$ exit
$else
$ primary_node = p2
$endif
$!
$! Get the node name of this node
$!
$this_node=f$getsyi("nodename")
$!
$! Start the Failover Monitor on this node
$!
$runmqfm -m 'qmgr_name'
$!
$! Check that the Failover Monitor has fully started
$! Wait up to 3 seconds
$!
$count = 0
$check_start:
$on error then continue
$!
$! Set the MQS$* symbols to the state of Failover Set
$! Wait up to 3 seconds
$!
```

Figure 24. start_failover_set command procedure (Part 1 of 2)

```
$failover -m 'qmgr_name' -l
$!
$! If an error is returned wait a second and try again
$!
$if ( ($status/8) .and %xfff ) .ne. 0 then goto wait
$!
$! If this node is not listed as running a monitor wait a second and try again
$!
$if f$locate( this_node, mqs$monitor_nodes ) .ne. f$length( mqs$monitor_nodes )
$then
$ goto start_qm
$endif
$wait:
$on error then exit
$count = count + 1
$!
$! If we have waited 3 seconds display an error and exit
$!
$if count .ge. 3
$then
$ write sys$output "Failover Monitor not started"
$ exit
$else
$ wait 00:00:01
$ goto check_start
$endif
$start_qm:
$!
$! Only start the Queue Manager on the primary node
$!
$if this_node .nes. primary_node
$then
$ write sys$output "Queue Manager not started on Secondary node"
$ exit
$endif
$!
$! Start the Queue Manager on the primary node if it is not already running.
$!
$if mqs$qmgr_node .eqs. ""
$then
$ failover -m 'qmgr_name' -n 'this_node' -s
$else
write sys$output "Queue Manager already started"
$endif
$exit
```

Figure 24. start_failover_set command procedure (Part 2 of 2)

Example failover set end command procedure, end_failover_set.com

The **end_failover_set.com** command procedure is used to conditionally end the queue manager and then the failover monitor on each node. The procedure is called from the site-specific shutdown before the **MQS_SHUTDOWN.COM** command procedure has been executed. The procedure is passed one parameter, the queue manager name. In this case it is called as follows:

```
$@start_failover_set testqm
```

The **end_failover_set.com** command procedure obtains the failover set state using the -l parameter of the **failover** command. Then if the queue manager is running on this node, it is ended. Then the failover monitor is halted.

```
on error then exit
$@sys$manager:mqs_symbols
$!
$! end_failover_set.com
$! --------------------------
$! Command procedure to end a Failover Set Queue Manager during shutdown
$!
$! p1 = Queue Manager name
$!
$! Check that the Queue Manager has been specified
$!
$if p1 .eqs ""
$then
$ Write sys$output "Queue Manager name omitted"
$ exit
$else
$ qmgr_name = p1
$endif
$!
$! Get the node name of this node
$!
$this_node=f$getsyi("nodename")
$!
$! Set the MQS$* symbols to the state of the Failover Set
$!
$failover -m 'qmgr_name' -l
$!
$! If an error then exit
$!
$if ( ($status/8) .and %xfff ) .ne. 0
$then
$ write sys$output "Error querying Failover Set"
$ exit
$endif
$!
$! If the Queue Manager is not running on this node then exit
$!
$ if mqs$qmgr_node .nes. this_node
$then
$ write sys$output "Queue Manager not running on this node"
$ goto halt_fm
$endif
$!
$! End the Queue Manager
$!
$failover -m gjtest -e
$halt_fm:
$!
$! Halt the Failover Monitor
$!
$failover -m gjtest -n 'this_node' -h
```

Figure 25. end_failover_set command procedure

Part 2. Reference

Chapter 17. MQSeries control commands

This chapter contains reference material for the control commands used with MQSeries for Compaq OpenVMS. All commands in this chapter can be issued from an OpenVMS DCL prompt.

Command names and their flags are not case sensitive: you can enter them in upper case, lower case, or a combination of upper case and lower case. However, parameters to control commands (such as queue names) can be case sensitive. See "Case sensitivity in control commands" on page 20 for more information.

Before using any control command, mqs_startup must have been run once since the last reboot.

Rules for naming MQSeries objects

In general, the names of MQSeries objects can have up to 48 characters. This rule applies to all the following objects:
- Queue managers (However, if the queue manager is supported by an OpenVMS cluster failover set then the maximum length is 25 characters.)
- Queues
- Process definitions
- Namelists
- Clusters

The maximum length of channel names is 20 characters.

The characters that can be used for all MQSeries names are:
- Upper case A–Z
- Lower case a–z
- Numerics 0–9
- Period (.)
- Underscore (_)
- Forward slash (/) (see note 1)
- Percent sign (%) (see note 1)

Notes:
1. Forward slash and percent are special characters. However, you cannot use forward slash and percent as the first character in a name. If you use either of these characters in a name, the name must be enclosed in double quotation marks whenever it is used.
2. Leading or embedded blanks are not allowed.
3. National language characters are not allowed.
4. Names may be enclosed in double quotation marks, but this is only essential if special characters are included in the name, or if case needs to be preserved.

Looking at object files

Each MQSeries queue, queue manager, or process object is represented by a file. Because these object names are not necessarily valid file names, the queue manager converts the object name into a valid file name, where necessary. This is described in "Understanding MQSeries file names" on page 19.

Names

To find out how to display the real file name of an object, see “dspmqfls (Display MQSeries files)” on page 244.

How to read syntax diagrams

This chapter contains syntax diagrams (sometimes referred to as “railroad” diagrams).

Each syntax diagram begins with a double right arrow and ends with a right and left arrow pair. Lines beginning with a single right arrow are continuation lines. You read a syntax diagram from left to right and from top to bottom, following the direction of the arrows.

Other conventions used in syntax diagrams are:

Table 15. How to read syntax diagrams

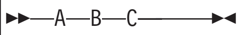
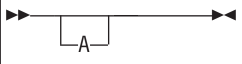
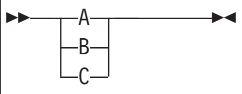
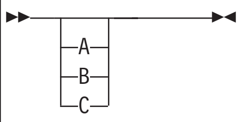
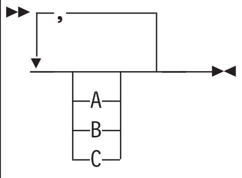
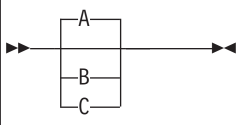
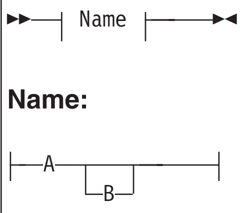
Convention	Meaning
	You must specify values A, B, and C. Required values are shown on the main line of a syntax diagram.
	You may specify value A. Optional values are shown below the main line of a syntax diagram.
	Values A, B, and C are alternatives, one of which you must specify.
	Values A, B, and C are alternatives, one of which you may specify.
	You may specify one or more of the values A, B, and C. Any required separator for multiple or repeated values (in this example, the comma (,)) is shown on the arrow.
	Values A, B, and C are alternatives, one of which you may specify. If you specify none of the values shown, the default A (the value shown above the main line) is used.

Table 15. How to read syntax diagrams (continued)

Convention	Meaning
	The syntax fragment Name is shown separately from the main syntax diagram.

Syntax help

You can obtain help about the syntax of any of the commands in this chapter by entering the command followed by a question mark. MQSeries responds by listing the syntax required for the selected command. The syntax shows all the parameters and variables associated with the command. Different forms of parentheses are used to indicate whether a parameter is required or not. For example:

```
CmdName [-x OptParam ] ( -c | -b ) { -p principal } argument
```

where:

CmdName	Is the command name for which help has been requested.
[-x OptParam]	The square brackets indicate that this is an optional parameter.
(-c | -b)	A mandatory field. In this case, you must select one of the flags c or b.
{ -p principal }	An optional list of variables that you may supply, but, if this is shown, at least one variable must be provided when you enter the command.
argument	An argument required to be supplied with this command, mandatory if shown on the response to the query.

Examples

1. Result of entering `endmqm ?`

```
endmqm [-z][-c | -i | -p] QMgrName
```

2. Result of entering `rcdmqimg ?`

```
rcdmqimg [-z] [-m QMgrName] -t ObjType [GenericObjName]
```

MQSeries return codes

Most of the MQSeries commands, for example **crtmqm**, write a status line when ending to indicate the success or failure of the command.

If the status of a command is to be tested in a DCL command file, it may be necessary to interpret the status value returned from an MQSeries program.

The MQSeries return codes are defined in a message file called `SYS$MESSAGE:MQS_MSG.EXE`.

To access the message text associated with a return code in the file, you ***must*** use the DCL SET MESSAGE command. This command loads the message codes into the message table of your process. For example:

```
$ SET MESSAGE SYS$MESSAGE:MQS_MSG.EXE
```

After this, you can use the `F$MESSAGE` lexical function to print the text of an MQSeries return code. For example:

```
$ strmqm )(*bad-qm-name&%$#
The queue manager name is either not valid or not known
$ WRITE SYS$OUTPUT F$MESSAGE($STATUS)
%MQS-F-CSPRC_Q_MGR_NAM, Queue manager name error
```

To convert the OpenVMS return code to a return code value used in MQSeries for OS/2 or UNIX systems, you can use the following DCL equation:

```
$ RC = $STATUS / 8 .AND. %xFFF
```

For example:

```
$ crtmqm &*)*(
The queue manager name is either not valid or not known
$ RC = $STATUS / 8 .AND. %xFFF
$ SHOW SYMBOL RC
  RC = 72   Hex = 00000048  Octal = 00000000110
```

crtmqcvx (Data conversion)

Purpose

Use the **crtmqcvx** command to create a fragment of code that performs data conversion on data type structures. The command generates a C function that can be used in an exit to convert your C structures.

The command reads an input file containing a structure or structures to be converted. It then writes an output file containing a code fragment or fragments to convert those structures.

For further information about this command and how to use it, refer to the *MQSeries Application Programming Guide*.

Syntax

```
►►──crtmqcvx──SourceFile──TargetFile──────────────────────────────────────►◄
```

Required parameters

SourceFile
: Specifies the input file containing the C structures to be converted.

TargetFile
: Specifies the output file containing the code fragments generated to convert the structures.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

The following example shows the results of using the data conversion command against a source C structure. The command issued is:

```
crtmqcvx source.tmp target.c
```

The input file, `source.tmp` looks like this:

```
/* This is a test C structure which can be converted by the */
/* crtmqcvx utility                                         */

struct my_structure
{
    int    code;
    MQLONG value;
};
```

The output file, `target.c`, produced by the command is shown below. You can use these code fragments in your applications to convert data structures. However, if you do so, you should understand that the fragment uses macros supplied in the MQSeries header file `amqsvmha.h`.

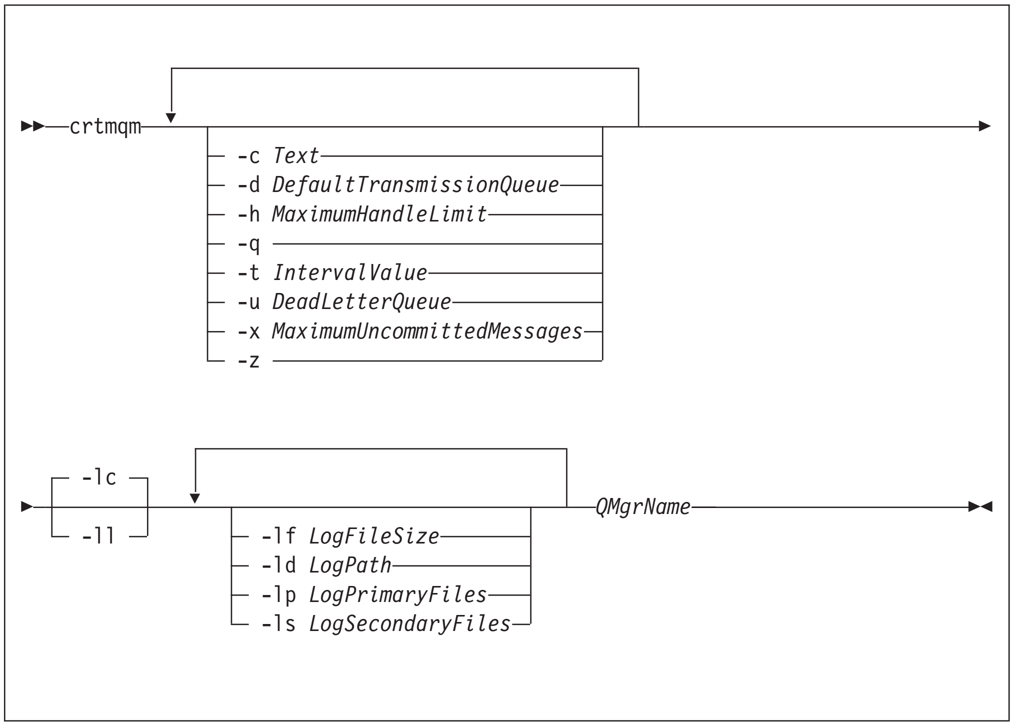
```
MQLONG Convertmy_structure(
             PMQBYTE *in_cursor,
             PMQBYTE *out_cursor,
             PMQBYTE in_lastbyte,
             PMQBYTE out_lastbyte,
             MQHCONN hConn,
             MQLONG  opts,
             MQLONG  MsgEncoding,
             MQLONG  ReqEncoding,
             MQLONG  MsgCCSID,
             MQLONG  ReqCCSID,
             MQLONG  CompCode,
             MQLONG  Reason)
{
    MQLONG ReturnCode = MQRC_NONE;

    ConvertLong(1); /* code */

    AlignLong();
    ConvertLong(1); /* value */

Fail:
    return(ReturnCode);
}
```

crtmqm (Create queue manager)

Purpose

Use the **crtmqm** command to create a local queue manager. Once a queue manager has been created, use the **strmqm** command to start it.

Syntax



Required parameters

QMgrName
: Specifies the name of the queue manager to be created. The name can contain up to 48 characters. This must be the last item in the command.

Optional parameters

-c *Text*
: Specify some descriptive text for this queue manager. The default is all blanks.

  You can use up to 64 characters. If mixed case is required, the description must be enclosed in double quotes.

-d *DefaultTransmissionQueue*
: Specifies the name of the local transmission queue that remote messages are placed on if a transmission queue is not explicitly defined for their destination. There is no default.

-h *MaximumHandleLimit*
: Specifies the maximum number of handles that any one application can have open at the same time.

  Specify a value in the range 1 through 999 999 999. The default value is 256.

-q Specifies that this queue manager is to be made the default queue manager. The new queue manager replaces any existing queue manager as the default.

If you accidentally use this flag and wish to revert to an existing queue manager as the default queue manager, you can edit the *DefaultQueueManager* stanza in the MQSeries configuration file. See “Chapter 13. Configuring MQSeries” on page 159 for information about configuration files.

-t *IntervalValue*
Specifies the trigger time interval in milliseconds for all queues controlled by this queue manager. This value specifies the time after the receipt of a trigger generating message when triggering is suspended. That is, if the arrival of a message on a queue causes a trigger message to be put on the initiation queue, any message arriving on the same queue within the specified interval does not generate another trigger message.

You can use the trigger time interval to ensure that your application is allowed sufficient time to deal with a trigger condition before it is alerted to deal with another on the same queue. You may wish to see all trigger events that happen; if so, set a low or zero value in this field.

Specify a value in the range 0 through 999 999 999. The default is 999 999 999 milliseconds, a time of more than 11 days. Allowing the default to be taken effectively means that triggering is disabled after the first trigger message. However, triggering can be reenabled by an application servicing the queue using an alter queue command to reset the trigger attribute.

-u *DeadLetterQueue*
Specifies the name of the local queue that is to be used as the dead-letter (undelivered-message) queue. Messages are put on this queue if they cannot be routed to their correct destination.

The default if the attribute is omitted is no dead-letter queue.

-x *MaximumUncommittedMessages*
Specifies the maximum number of uncommitted messages under any one syncpoint. That is, the sum of:
- The number of messages that can be retrieved from queues
- The number of messages that can be put on queues
- Any trigger messages generated within this unit of work

This limit does not apply to messages that are retrieved or put outside a syncpoint.

Specify a value in the range 1 through 10 000. The default value is 1000 uncommitted messages.

-z Suppresses error messages.

This flag is normally used within MQSeries to suppress unwanted error messages. As use of this flag could result in loss of information, it is recommended that you do not use it when entering commands on a command line.

The following set of flags is used to define the logging to be used by the queue manager being created. For more information about logs, see “Using the log for recovery” on page 133.

-lc Circular logging is to be used. This is the default logging method.

-ll Linear logging is to be used.

-lf *LogFileSize*
: Specifies the size of the log files in units of 4 KB. The minimum value is 64, and the maximum is 16384 The default value is 1024, giving a default log size of 4 MB.

-ld *LogPath*
: Specifies the directory to be used to hold the log files. The default is `MQS_ROOT:[MQM.LOG]`. The default can also be changed when MQSeries is customized.

  User ID mqm and group mqm must have full authorities to the log files. If you change the locations of these files, you must give these authorities yourself. This is done automatically if the logs files are in their default locations.

-lp *LogPrimaryFiles*
: Specifies the number of primary log files to be allocated. The default value is 3, the minimum is 2, and the maximum is 62.

-ls *LogSecondaryFiles*
: Specifies the number of secondary log files to be allocated. The default value is 2, the minimum is 1, and the maximum is 61.

  Note: The total number of log files is restricted to 63, regardless of the number requested.

  The limits given in the previous parameter descriptions are limits set by MQSeries. Operating system limits may reduce the maximum possible log size.

Return codes

0	Queue manager created
8	Queue manager already exists
49	Queue manager stopping
69	Storage not available
70	Queue space not available
71	Unexpected error
72	Queue manager name error
100	Log location invalid
111	Queue manager created. However, there was a problem processing the default queue manager definition in the product configuration file. The default queue manager specification may be incorrect.
115	Invalid log size

Examples

1. This command creates a default queue manager named `Paint.queue.manager`, which is given a description of `Paint shop`. It also specifies that linear logging is to be used:

```
crtmqm -c "Paint shop" -ll -q "Paint.queue.manager"
```

2. This example requests a number of log files. Two primary and three secondary log files are specified.

```
crtmqm -c "Paint shop" -ll -lp 2 -ls 3 -q "Paint.queue.manager"
```

3. In this example, another queue manager, `travel`, is created. The trigger interval is defined as 5000 milliseconds (or 5 seconds) and its dead-letter queue is specified as SYSTEM.DEAD.LETTER.QUEUE.

```
crtmqm -t 5000 -u SYSTEM.DEAD.LETTER.QUEUE "travel"
```

   Once a trigger event has been generated, further trigger events are disabled for five seconds.

Related commands

strmqm	Start queue manager
endmqm	End queue manager
dltmqm	Delete queue manager

dltmqm (Delete queue manager)

Purpose

Use the **dltmqm** command to delete a specified queue manager. All objects associated with this queue manager are also deleted. Before you can delete a queue manager you must end it using the **endmqm** command.

Syntax

```
►►──dltmqm──┬──────┬──QMgrName──────────────────────────────────────────►◄
            └─ -z ─┘
```

Required parameters

QMgrName
: Specifies the name of the queue manager to be deleted.

Optional parameters

-z Suppresses error messages.

Return codes

0	Queue manager deleted
3	Queue manager being created
5	Queue manager running
16	Queue manager does not exist
49	Queue manager stopping
69	Storage not available
71	Unexpected error
72	Queue manager name error
100	Log location invalid
112	Queue manager deleted. However, there was a problem processing the default queue manager definition in the product configuration file. The default queue manager specification may be incorrect.

Examples

1. The following command deletes the queue manager `saturn.queue.manager`.

```
dltmqm "saturn.queue.manager"
```

2. The following command deletes the queue manager `travel` and also suppresses any messages caused by the command.

```
dltmqm -z "travel"
```

Related commands

crtmqm	Create queue manager
strmqm	Start queue manager
endmqm	End queue manager

dmpmqlog (Dump log)

Purpose

Use the **dmpmqlog** command to dump a formatted version of the MQSeries system log.

The log to be dumped must have been created on the same type of operating system as that being used to issue the command.

Syntax

```
►►──dmpmqlog──┬───────────────────┬──┬─────────────┬──┬────────────────┬──────────►
              ├─ -b ──────────────┤  └─ -e EndLSN──┘  └─ -f LogFilePath─┘
              ├─ -s StartLSN──────┤
              └─ -n ExtentNumber──┘

►──┬──────────────┬───────────────────────────────────────────────────────────────►◄
   └─ -m QMgrName─┘
```

Optional parameters

Dump start point
Use one of the following parameters to specify the log sequence number (LSN) at which the dump should start. If no start point is specified, dumping starts by default from the LSN of the first record in the active portion of the log.

-b Specifies that dumping should start from the base LSN. The base LSN identifies the start of the log extent that contains the start of the active portion of the log.

-s *StartLSN*
Specifies that dumping should start from the specified LSN. The LSN is specified in the format `nnnn:nnnn:nnnn:nnnn`.

If you are using a circular log, the LSN value must be equal to or greater than the base LSN value of the log.

-n *ExtentNumber*
Specifies that dumping should start from the specified extent number. The extent number must be in the range 0–9 999 999.

This parameter is valid only for queue managers whose *LogType* (as recorded in the configuration file, qm.ini) is LINEAR.

-e *EndLSN*
Specifies that dumping should end at the specified LSN. The LSN is specified in the format `nnnn:nnnn:nnnn:nnnn`.

-f *LogFilePath*
Is the absolute, rather than the relative, directory path name to the log files. The specified directory must contain the log header file (`amqhlctl.lfh`) and a subdirectory called `active`. The active subdirectory must contain the log files. By default, log files are assumed to be in the directories specified in the

`mqs.ini` and `qm.ini` files. If this option is used then queue names, associated with queue identifiers, will only be shown in the dump if a queue manager name is specified explicitly for the -m option and that queue manager has the object catalog file in its directory path.

On a system that supports long filenames this file is named `qmqmobjcat` and, in order to map the queue identifiers to queue names, it must be the file used when the log files were created. As an example, for a queue manager named qm1, the object catalog file is located in the directory `MQS_ROOT:[MQM.QMGRS.QM1.QMANAGER]`. To achieve this mapping, it may be necessary to create a temporary queue manager, for example named tmpq, replace its object catalog with the one associated with the specific log files, and then start dmpmqlog, specifying `-m tmpq` and `-f` with the absolute directory path name to the log files.

-m *QMgrName*
Is the name of the queue manager. If this parameter is omitted, the name of the default queue manager is used.

The queue manager you specify, or default to, must not be running when the **dmpmqlog** command is issued. Similarly, the queue manager must not be started while **dmpmqlog** is running.

dspmqaut (Display authority)

Purpose

Use the **dspmqaut** command to display the current authorizations to a specified object.

Only one group may be specified.

If a user ID is a member of more than one group, this command displays the combined authorizations of all of the groups.

Syntax

```
►►──dspmqaut──┬────────────────┬── -n ObjectName── -t ObjectType──────────────►
              └─ -m QMgrName──┘

►──┬─ -g GroupName─────┬──┬──────────────────────────┬──────────────────────►◄
   └─ -p PrincipalName─┘  └─ -s ServiceComponent────┘
```

Required parameters

-n *ObjectName*
: Specifies the name of the object on which the inquiry is to be made.

  This is a required parameter ***unless*** the inquiry relates to the queue manager itself, in which case it must not be included.

  You must specify the name of a queue manager, queue, or process definition.

-t *ObjectType*
: Specifies the type of object on which the inquiry is to be made. Possible values are:

queue or q	A queue or queues matching the object type parameter
qmgr	A queue manager object
process or prcs	A process
namelist or nl	A namelist

Optional parameters

-m *QMgrName*
: Specifies the name of the queue manager on which the inquiry is to be made.

-g *GroupName*
: Specifies the name of the user group on which the inquiry is to be made. You can only specify ***one*** name, which must be the name of an existing rights identifier.

-p *PrincipalName*
: Specifies the name of a user whose authorizations to the specified object are to be displayed.

-s *ServiceComponent*
This parameter only applies if you are using installable authorization services, otherwise it is ignored.

If installable authorization services are supported, this parameter specifies the name of the authorization service to which the authorizations apply. This parameter is optional; if it is not specified, the authorization inquiry is made to the first installable component for the service.

Returned parameters

This command returns an authorization list, which can contain none, one, or more authorization parameters. Each authorization parameter returned means that any user ID in the specified group has the authority to perform the operation defined by that parameter.

Table 16 shows the authorities that can be given to the different object types.

Table 16. Security authorities from the dspmqaut command

Authority	Queue	Process	Qmgr	Namelist
all	✓	✓	✓	✓
alladm	✓	✓	✓	✓
allmqi	✓	✓	✓	✓
altusr			✓	
browse	✓			
chg	✓	✓	✓	✓
clr	✓			
connect			✓	
cpy	✓	✓	✓	✓
crt	✓	✓	✓	✓
dlt	✓	✓	✓	✓
dsp	✓	✓	✓	✓
get	✓			
inq	✓	✓	✓	✓
passall	✓			
passid	✓			
put	✓			
set	✓	✓	✓	
setall	✓		✓	
setid	✓		✓	

The following list defines the authorizations associated with each parameter:

all	Use all operations relevant to the object.
alladm	Perform all administration operations relevant to the object.
allmqi	Use all MQI calls relevant to the object.
altusr	Specify an alternate user ID on an MQI call.

browse	Retrieve a message from a queue by issuing an MQGET call with the BROWSE option.
chg	Change the attributes of the specified object, using the appropriate command set.
clr	Clear a queue (PCF command Clear queue only).
connect	Connect the application to the specified queue manager by issuing an MQCONN call.
cpy	Copy the definition of an object, for example, the PCF Copy queue command.
crt	Create objects of the specified type, using the appropriate command set.
dlt	Delete the specified object, using the appropriate command set.
dsp	Display the attributes of the specified object, using the appropriate command set.
get	Retrieve a message from a queue by issuing an MQGET call.
inq	Make an inquiry on a specific queue by issuing an MQINQ call.
passall	Pass all context.
passid	Pass the identity context.
put	Put a message on a specific queue by issuing an MQPUT call.
set	Set attributes on a queue from the MQI by issuing an MQSET call.
setall	Set all context on a queue.
setid	Set the identity context on a queue.

The authorizations for administration operations, where supported, apply to these command sets:
- Control commands
- MQSC commands
- PCF commands

Return codes

0	Successful operation
36	Invalid arguments supplied
40	Queue manager not available
49	Queue manager stopping
69	Storage not available
71	Unexpected error
72	Queue manager name error
133	Unknown object name
145	Unexpected object name
146	Object name missing
147	Object type missing
148	Invalid object type
149	Entity name missing

Examples

The following example shows a command to display the authorizations on queue manager `saturn.queue.manager` associated with user group `staff`:

```
dspmqaut -m "saturn.queue.manager" -t qmgr -g staff
```

The results from this command are:

```


Entity staff has the following authorizations for object:
        get
        browse
        put
        inq
        set
        connect
        altusr
        passid
        passall
        setid
```

Related commands

setmqaut	Set or reset authority

dspmqcsv (Display command server)

Purpose

Use the **dspmqcsv** command to display the status of the command server for the specified queue manager.

The status can be one of the following:
- Starting
- Running
- Running with SYSTEM.ADMIN.COMMAND.QUEUE not enabled for gets
- Ending
- Stopped

Syntax

```
►►──dspmqcsv──┬───────────┬────────────────────────────────────────────►◄
              └─QMgrName──┘
```

Optional parameters

QMgrName
: Specifies the name of the local queue manager for which the command server status is being requested.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

The following command displays the status of the command server associated with `venus.q.mgr`:

```
dspmqcsv "venus.q.mgr"
```

Related commands

strmqcsv	Start a command server
endmqcsv	End a command server

dspmqfls (Display MQSeries files)

Purpose

Use the **dspmqfls** command to display the real file system name for all MQSeries objects that match a specified criterion. You can use this command to identify the files associated with a particular MQSeries object. This is useful for backing up specific objects. See “Understanding MQSeries file names” on page 19 for further information about name transformation.

Syntax

```
►►──dspmqfls──┬──────────────┬──┬────────────┬──GenericObjName──────────────►◄
              └─ -m QMgrName─┘  └─ -t ObjType─┘
```

Required parameters

GenericObjName
: Specifies the name of the MQSeries object. The name is a string with no flag and is a required parameter. If the name is omitted an error is returned.

  This parameter supports a wild card character * at the end of the string.

Optional parameters

-m *QMgrName*
: Specifies the name of the queue manager for which files are to be examined. If omitted, the command operates on the default queue manager.

-t *ObjType*
: Specifies the MQSeries object type. The following list shows the valid object types. The abbreviated name is shown first followed by the full name.

*** or all**	All object types; this is the default
q or queue	A queue or queues matching the object name parameter
ql or qlocal	A local queue
qa or qalias	An alias queue
qr or qremote	A remote queue
qm or qmodel	A model queue
qmgr	A queue manager object
prcs or process	A process
ctlg or catalog	An object catalog
nl or namelist	A namelist

Note: The **dspmqfls** command displays the directory containing the queue, ***not*** the name of the queue itself.

Return codes

0	Command completed normally
10	Command completed but not entirely as expected
20	An error occurred during processing

Examples

1. The following command displays the details of all objects with names beginning SYSTEM.ADMIN that are defined on the default queue manager.

```
dspmqfls SYSTEM.ADMIN*
```

2. The following command displays file details for all processes with names beginning PROC defined on queue manager RADIUS.

```
dspmqfls -m RADIUS -t prcs PROC*
```

dspmqtrc (Display MQSeries formatted trace output)

Purpose

Use the **dspmqtrc** command to display MQSeries formatted trace output.

Syntax

```
►►──dspmqtrc──┬──────────────────┬──┬─────┬──┬──────────────────────┬─────────►
              └─ -t FormatTemplate┘  └─ -h ┘  └─ -o OutputFilename──┘

►─InputFileName────────────────────────────────────────────────────────────►◄
```

Required parameters

InputFileName
: Specifies the name of the file containing the unformatted trace. For example `MQS_ROOT:[MQM.TRACE]AMQ20202345.TRC`.

Optional parameters

-t *FormatTemplate*
: Specifies the name of the template file containing details of how to display the trace. The default value is `SYS$SHARE:AMQTRC.FMT`.

-h Omit header information from the report.

-o *output_filename*
: The name of the file into which to write formatted data.

Examples

1. The following command shows the redirection of output:

```
dspmqtrc mqs_root:[mqm.trace]amq20202345.trc > mqs_root:[mqm.trace]amq20202345.fmt
```

Related commands

endmqtrc	End MQSeries trace
strmqtrc	Start MQSeries trace

dspmqtrn (Display MQSeries transactions)

Purpose

Use the **dspmqtrn** command to display the transactions that are in prepared status in a two-phase commit procedure and that are known to a queue manager (see the Attention notice below).

Each transaction is displayed as a transaction number (a human-readable transaction identifier), the transaction state, and the transaction ID. Transaction IDs can be up to 128 characters long, hence the need for a transaction number.

Syntax

```
►►──dspmqtrn──┬──────┬──┬──────┬──┬───────────────┬──────────────────────►◄
              └─ -e ─┘  └─ -i ─┘  └─ -m  QMgrName─┘
```

Attention: The only time that you can expect to use this command is if you are using an external transaction manager and are involved with two-phase commitment procedures. If you do not use two-phase commit, do not use this command. This command should be used only if the syncpoint manager has failed to resolve a transaction.

Optional parameters

-m *QMgrName*
Specifies the name of the queue manager whose transactions are to be examined. If omitted, the command operates on the default queue manager.

-e Requests details of externally coordinated, in-doubt transactions. Such transactions are those for which MQSeries has been asked to prepare to commit, but has not yet been informed of the transaction outcome.

-i Requests details of internally coordinated, in-doubt transactions. Such transactions are those for each resource manager has been asked to prepare to commit, but MQSeries has yet to inform the resource managers of the transaction outcome.

Information about the deduced state of the transaction in each of its participating resource managers is displayed. This information can help you assess the effects of failure in a particular resource manager.

Note: If you specify neither -e or -i, details of both internally and externally coordinated in-doubt transactions are displayed.

Return codes

0	Successful operation
36	Invalid arguments supplied
40	Queue manager not available
49	Queue manager stopping
69	Storage not available
71	Unexpected error

72	Queue manager name error
102	No transactions found

Related commands

rsvmqtrn	Resolve MQSeries transaction

endmqcsv (End command server)

Purpose

Use the **endmqcsv** command to stop the command server on the specified queue manager.

Syntax

```
             ┌─ -c ─┐
►►──endmqcsv─┼──────┼─QMgrName──────────────────────────────────►◄
             └─ -i ─┘
```

Required parameters

QMgrName
: Specifies the name of the queue manager for which the command server is to be ended.

Optional parameters

-c Specifies that the command server is to be stopped in a controlled manner. The command server is allowed to complete the processing of any command message that it has already started. No new message is read from the command queue.

This is the default.

-i Specifies that the command server is to be stopped immediately. Actions associated with a command message currently being processed may not be completed.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

1. The following command stops the command server on queue manager `saturn.queue.manager`:

```
endmqcsv  -c "saturn.queue.manager"
```

   The command server can complete processing any command it has already started before it stops. Any new commands received remain unprocessed in the command queue until the command server is restarted.

2. The following command stops the command server on queue manager `pluto` immediately:

```
endmqcsv -i "pluto"
```

Related commands

strmqcsv	Start a command server
dspmqcsv	Display the status of a command server

endmqlsr (End listener)

Purpose

The **endmqlsr** command ends all listener process for the specified queue manager.

The queue manager must be stopped before the **endmqlsr** command is issued.

Syntax

```
►►──endmqlsr──┬────────────────┬──────────────────────────────────────────►◄
              └─ -m QMgrName ──┘
```

Optional parameters

-m *QMgrName*
: Specifies the name of the queue manager. If no name is specified, the processing will be done for the default queue manager.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

endmqm (End queue manager)

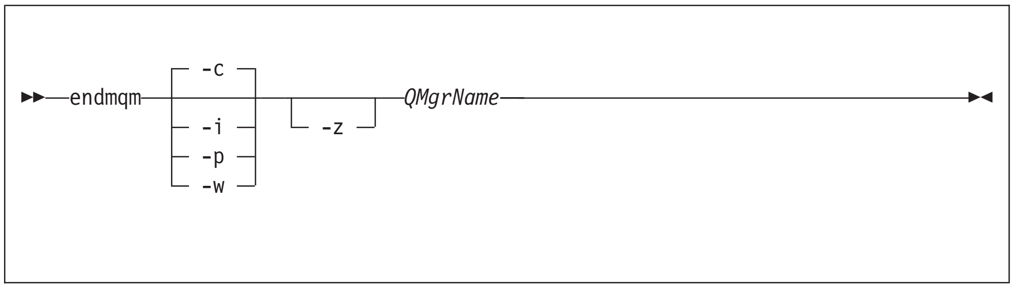
Purpose

Use the **endmqm** command to end (stop) a specified local queue manager. This command stops a queue manager in one of three modes:
- Normal or quiesced shutdown
- Immediate shutdown
- Preemptive shutdown

The attributes of the queue manager and the objects associated with it are not affected. You can restart the queue manager using the **strmqm** (Start queue manager) command.

To delete a queue manager, you must stop it and then use the **dltmqm** (Delete queue manager) command.

Syntax



Required parameters

QMgrName
: Specifies the name of the message queue manager to be stopped.

Optional parameters

-c Controlled (or quiesced) shutdown. The queue manager stops but only after all applications have disconnected. Any MQI calls currently being processed are completed. This is the default.

Control is returned to you immediately and you are not notified of when the queue manager has stopped.

-w Wait shutdown

This type of shutdown is equivalent to a controlled shutdown except that control is returned to you only after the queue manager has stopped. You receive the message "Waiting for queue manager *QMgrName* to end" while shutdown progresses.

-i Immediate shutdown. The queue manager stops after it has completed all the MQI calls currently being processed. Any MQI requests issued after the command has been issued fail. Any incomplete units of work are rolled back when the queue manager is next started.

-p Preemptive shutdown.

Use this type of shutdown only in exceptional circumstances. For example, when a queue manager does not stop as a result of a normal **endmqm** command.

The queue manager stops without waiting for applications to disconnect or for MQI calls to complete. This can give unpredictable results for MQSeries applications. All processes in the queue manager that fail to stop are terminated 30 seconds after the command is issued.

Note: After a forced or preemptive shutdown, or if the queue manager fails, the queue manager may have ended without cleaning up the shared memory that it owns. This can lead to problems restarting. For information on how to use the MONMQ utility to clean up after an abrupt ending of this type, see "Managing shared memory with MONMQ" on page 342.

-z Suppresses error messages on the command.

Return codes

0	Queue manager ended
3	Queue manager being created
16	Queue manager does not exist
40	Queue manager not available
49	Queue manager stopping
69	Storage not available
71	Unexpected error
72	Queue manager name error

Examples

The following examples show commands that end (stop) the specified queue managers.

1. This command ends the queue manager named `mercury.queue.manager` in a controlled way. All applications currently connected are allowed to disconnect.

```
endmqm "mercury.queue.manager"
```

2. This command ends the queue manager named `saturn.queue.manager` immediately. All current MQI calls complete, but no new ones are allowed.

```
endmqm -i "saturn.queue.manager"
```

Related commands

crtmqm	Create a queue manager
strmqm	Start a queue manager
dltmqm	Delete a queue manager

endmqtrc (End MQSeries trace)

Purpose

Use the **endmqtrc** command to end tracing for the specified entity or all entities.

Syntax

```
►►──endmqtrc──┬──────────────────────┬────────────────────────────────────►◄
              ├─ -a ─────────────────┤
              └─ -m QMgrName──┬──────┤
                              └─ -e ─┘
```

Optional parameters

-m *QMgrName*
Is the name of the queue manager for which tracing is to be ended.

A maximum of one -m flag and associated queue manager name can be supplied on the command.

A queue manager name and -m flag can be specified on the same command as the -e flag.

-e If this flag is specified, early tracing is ended.

-a If this flag is specified all tracing is ended.

This flag ***must*** be specified alone.

Return codes

AMQ5611	This message is issued if arguments that are not valid are supplied to the command.

Examples

This command ends tracing of data for a queue manager called `QM1`.

```
endmqtrc -m QM1
```

Related commands

dspmqtrc	Display formatted trace output
strmqtrc	Start MQSeries trace

failover (Manage a failover set)

Purpose

Use the **failover** command to manage a failover set. The **failover** command includes both update and query parameters. The **failover** command can be executed from any OpenVMS node in the failover set.

Syntax

```
►►──failover── -m QMgrName──┬──────────────┬──────────────────────────────────►
                            └─ -n NodeName─┘

►─┬─────────────────────────────────────────────────────────────────────┬───►◄
  ├─ -q ────────────────────────────────────────────────────────────────┤
  ├─ -l ────────────────────────────────────────────────────────────────┤
  ├─ -s ────────────────────────────────────────────────────────────────┤
  ├─ -e ────────────────────────────────────────────────────────────────┤
  ├─ -f ────────────────────────────────────────────────────────────────┤
  ├─ -h ────────────────────────────────────────────────────────────────┤
  ├─ -u ────────────────────────────────────────────────────────────────┤
  └─ -c ──┬─────────────────────┬──┬──────────────────┬──┬─────────────────────┬─┘
          └─-cluster──state─────┘  └─-qmgr──state─────┘  └─-monitor──state─────┘
```

Required parameters

-m *QMgrName*
: Specifies the name of the queue manager to which the **failover** command is to be applied. The maximum length supported for the *QMgrName* name is 25 characters.

-n *NodeName*
: Specifies the OpenVMS node name to which the command applies. This parameter is required for the -h and the -c parameters.

Optional parameters

-q Queries the state of the failover set and displays the output.

-l Queries the state of the failover set and sets the following DCL symbols:

DCL symbol name	Description
MQS$QMGR_NODE	Set to the OpenVMS node that is running the queue manager and a null string if there is no queue manager running
MQS$AVAILABLE_NODES	Set to the list of OpenVMS nodes that are available to run the queue manager. That is the nodes that are in the queue manager AVAILABLE state and that have a failover monitor running.
MQS$MONITOR_NODES	Set to the list of OpenVMS nodes that have a failover monitor running on them

-s Starts the queue manager in the failover set. If the -n parameter is specified, the queue manager is started on the OpenVMS node specified; otherwise it is started on the highest priority available node.

-e Ends the queue manager in the failover set.

-f Moves the queue manager to another node in the failover set. If the -n parameter is specified, the queue manager is moved to the node specified; otherwise the queue manager is moved to the highest priority available node.

-h Halts the failover monitor that is running on the node specified with the -n parameter.

-u Clears the update in progress flag.

-c Changes the failover set state. The states that are changed are determined by the following three parameters. Changes take effect only if they are consistent with the running state of the failover set.

-cluster started | stopped
Used with the -c parameter to change the overall failover set state.

-qmgr available | running | excluded
Used with the -c parameter to change the node queue manager state for the node specified with the -n parameter.

-monitor started | stopped | watching
Used with the -c parameter to change the node failover monitor state for the node specified with the -n parameter.

Return codes

0	Command completed normally
5	The queue manager is running
36	Arguments supplied to a command are not valid
326	MQseries queue manager not running
1925	There is no failover monitor started for queue manager
1926	Failover set update operation in progress
1937	No node available on which to start the queue manager
1939	The ending of the queue manager was forced
1940	The ending of the queue manager timed out before completion.

OpenVMS error codes

36	%SYSTEM-F-NOPRIV, insufficient privilege or object protection violation
652	%SYSTEM-F-NOSUCHNODE, remote node is unknown
660	%SYSTEM-F-REJECT, connect to network object rejected

Examples

1. This example queries the state of a failover set for a queue manager called `testqm`.

```
failover -m "testqm" -q
```

2. This example starts the queue manager called `testqm` on node batman.

```
failover -m "testqm" -n batman -s
```

3. This example moves the queue manager called `testqm` to the highest priority available node.

```
failover -m "testqm" -f
```

Related commands

runmqfm	Start a failover monitor

rcdmqimg (Record media image)

Purpose

Use the **rcdmqimg** command to write an image of an MQSeries object, or group of objects, to the log for use in media recovery. Use the associated command **rcrmqobj** to recreate the object from the image.

This command is used with an active queue manager. Further activity on the queue manager is logged so that, although the image becomes out of date, the log records reflect any changes to the object.

Syntax

```
►►──rcdmqimg──┬────────────────┬──┬──────┬──-t ObjectType──GenericObjName──────────►◄
              └─ -m QMgrName ──┘  └─ -z ─┘
```

Required parameters

GenericObjName
: Specifies the name of the object that is to be recorded. This parameter may have a trailing asterisk to indicate that any objects with names matching the portion of the name prior to the asterisk are to be recorded.

  This parameter is required *unless* you are recording a queue manager object or the channel synchronization file. If you specify an object name for the channel synchronization file, it is ignored.

-t *ObjectType*
: Specifies the type of objects whose images are to be recorded. Valid object types are:

prcs or process	Processes
q or queue	All types of queue
ql or qlocal	Local queues
qa or qalias	Alias queues
qr or qremote	Remote queues
qm or qmodel	Model queues
qmgr	Queue manager object
syncfile	Channel synchronization file
nl or namelist	Namelists
ctlg or catalog	An object catalog
*** or all**	All of the above

Optional parameters

-m *QMgrName*
: Specifies the name of the queue manager for which images are to be recorded. If omitted, the command operates on the default queue manager.

-z Suppresses error messages.

Return codes

0	Successful operation
36	Invalid arguments supplied
40	Queue manager not available
49	Queue manager stopping
68	Media recovery is not supported
69	Storage not available
71	Unexpected error
72	Queue manager name error
119	User not authorized
128	No objects processed
131	Resource problem
132	Object damaged
135	Temporary object cannot be recorded

Examples

The following command records an image of the queue manager object `saturn.queue.manager` in the log.

```
rcdmqimg -t qmgr -m "saturn.queue.manager"
```

Related commands

rcrmqobj	Recreate a queue manager object

rcrmqobj (Recreate object)

Purpose

Use the **rcrmqobj** command to recreate an object, or group of objects, from their images contained in the log. Use the associated command, **rcdmqimg**, to record the object images to the log.

This command must be used on a running queue manager. All activity on the queue manager after the image was recorded is logged. To recreate an object you must replay the log to recreate events that occurred after the object image was captured.

Syntax

```
►►──rcrmqobj──┬───────────────┬──┬──────┬──-t ObjectType──GenericObjName──────►◄
              └─-m QMgrName───┘  └─ -z ─┘
```

Required parameters

GenericObjName
: Specifies the name of the object that is to be recreated. This parameter may have a trailing asterisk to indicate that any objects with names matching the portion of the name prior to the asterisk are to be recreated.

  This parameter is required *unless* the object type is the channel synchronization file; if an object name is supplied for this object type, it is ignored.

-t *ObjectType*
: Specifies the type of objects to be recreated. Valid object types are:

prcs or process	Processes
q or queue	All types of queue
ql or qlocal	Local queues
qa or qalias	Alias queues
qr or qremote	Remote queues
qm or qmodel	Model queues
nl or namelist	Namelists
ctlg or catalog	An object catalog
*** or all**	All the above
syncfile	The channel synchronization file

Note: Using this flag causes the channel synchronization file to be regenerated for the queue manager specified. This is necessary because the file is not saved by the **rcdmqimg** command.

Optional parameters

-m *QMgrName*
Specifies the name of the queue manager for which objects are to be recreated. If omitted, the command operates on the default queue manager.

-z Suppresses error messages.

Return codes

0	Successful operation
36	Invalid arguments supplied
40	Queue manager not available
49	Queue manager stopping
66	Media image not available
68	Media recovery is not supported
69	Storage not available
71	Unexpected error
72	Queue manager name error
119	User not authorized
128	No objects processed
135	Temporary object cannot be recovered
136	Object in use

Examples

1. The following command recreates all local queues for the default queue manager:

```
rcrmqobj -t ql *
```

2. The following command recreates all remote queues associated with queue manager `store`:

```
rcrmqobj -m "store" -t qr *
```

Related commands

rcdmqimg	Record an MQSeries object in the log

rsvmqtrn (Resolve MQSeries transactions)

Purpose

Use the **rsvmqtrn** command to commit or back out internally or externally coordinated in-doubt transactions.

Use this command only when you are certain that transactions cannot be resolved by the normal protocols. Issuing this command may result in the loss of transactional integrity between resource managers for a distributed transaction.

Syntax

```
►►──rsvmqtrn──┬─ -a ──────────────────┬── -m QMgrName──────────────────────────►◄
              ├─ -b Transaction───────┤
              ├─ -c Transaction───────┤
              └─ -r RMIdTransaction───┘
```

Required parameters

-m *QMgrName*
: Specifies the name of the queue manager. This parameter is mandatory.

Optional parameters

-a Specifies that the queue manager should attempt to resolve all internally coordinated, in-doubt transactions (that is, all global units of work).

-b Specifies that the named transaction is to be backed out. This flag is valid for externally coordinated transactions (that is, for external units of work) only.

-c Specifies that the named transaction is to be committed. This flag is valid for externally coordinated transactions (that is, for external units of work) only.

-r *RMId*
: Identifies the resource manager to which the commit or back out decision applies. This flag is valid for internally coordinated transactions only, and for resource managers that are no longer configured in the queue manager's `qm.ini` file. The outcome delivered will be consistent with the decision reached by MQSeries for the transaction.

Transaction
: Is the transaction number of the transaction being committed or backed out. To discover the relevant transaction number, use the **dspmqtrn** command This parameter is required with the -b, -c, and -r *RMId* parameters.

Return codes

0	Successful operation
32	Transactions could not be resolved
34	Resource manager not recognized
35	Resource manager not permanently available
36	Invalid arguments supplied
40	Queue manager not available

49	Queue manager stopping
69	Storage not available
71	Unexpected error
72	Queue manager name error
85	Transactions not known

Related commands

dspmqtrn	Display list of prepared transactions

runmqchi (Run channel initiator)

Purpose

Use the **runmqchi** command to run a channel initiator process. For more information about the use of this command, refer to the *MQSeries Intercommunication* book.

Syntax

```
►►──runmqchi──┬────────────────────┬──┬───────────────┬─────────────────────►◄
              └─ -q InitiationQName─┘  └─ -m QMgrName─┘
```

Optional parameters

-q *InitiationQName*
: Specifies the name of the initiation queue to be processed by this channel initiator. If not specified, SYSTEM.CHANNEL.INITQ is used.

-m *QMgrName*
: Specifies the name of the queue manager on which the initiation queue exists. If the name is omitted, the default queue manager is used.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

If errors occur that result in return codes of either 10 or 20, you should review the queue manager error log that the channel is associated with for the error messages. You should also review the $SYSTEM error log, as problems that occur before the channel is associated with the queue manager are recorded there. For more information about error logs, see “Error logs” on page 187.

runmqchl (Run channel)

Purpose

Use the **runmqchl** command to run a Sender (SDR), a Requester (RQSTR).

The channel runs synchronously. To stop the channel, issue the MQSC command STOP CHANNEL.

Syntax

```
►►──runmqchl── -c ChannelName──┬──────────────┬──────────────────────►◄
                               └─ -m QMgrName─┘
```

Required parameters

-c *ChannelName*
Specifies the name of the channel to run.

Optional parameters

-m *QMgrName*
Specifies the name of the queue manager with which this channel is associated. If no name is specified, the default queue manager is used.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

If return codes 10 or 20 are generated, review the error log of the associated queue manager for the error messages. You should also review the $SYSTEM error log because problems that occur before the channel is associated with the queue manager are recorded there.

runmqdlq (Run dead-letter queue handler)

Purpose

Use the **runmqdlq** command to start the dead-letter queue (DLQ) handler, a utility that you can run to monitor and handle messages on a dead-letter queue.

The dead-letter queue handler can be used to perform various actions on selected messages by specifying a set of rules that can both select a message and define the action to be performed on that message.

The **runmqdlq** command takes its input from `SYS$INPUT` When the command is processed, the results and a summary are put into a report that is sent to `SYS$OUTPUT`.

By taking `SYS$INPUT` from the keyboard, you can enter **runmqdlq** rules interactively.

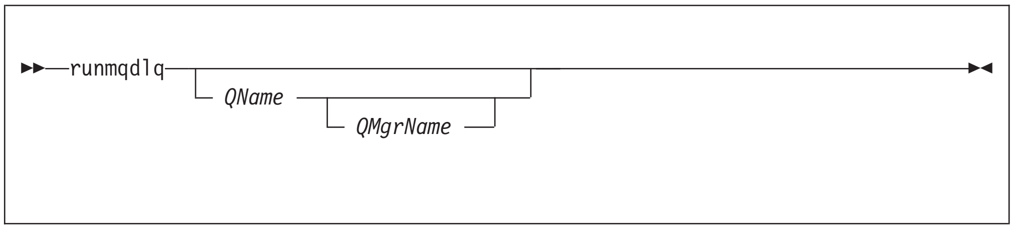
By redirecting the input from a file, you can apply a rules table to the specified queue. The rules table must contain at least one rule.

If the DLQ handler is used in foreground mode without redirecting `SYS$INPUT` from a file, (the rules table) the DLQ handler:

- Reads its input from the keyboard.
- Does not start to process the named queue until it receives an end_of_file (ctrl-Z) character.

For more information about rules tables and how to construct them, see "The DLQ handler rules table" on page 94.

Syntax



Optional parameters

The MQSC rules for comment lines and for joining lines also apply to the DLQ handler input parameters.

QName
: Specifies the name of the queue to be processed.

  If no name is specified the dead letter queue defined for the local queue manager is used. If one or more blanks (' ') are used, the dead letter queue of the local queue manager is explicitly assigned.

  A DLQ handler can be used to select particular messages on a dead-letter queue for special processing. For example, you could redirect the messages to different dead-letter queues. Subsequent processing with another instance of the DLQ handler might then process the messages, according to a different rules table.

QMgrName

The name of the queue manager that owns the queue to be processed.

If no name is specified, the default queue manager for the installation is used. If one or more blanks (' ') are used, the default queue manager for this installation is explicitly assigned.

runmqfm (Start a failover monitor)

Purpose

Use the **runmqfm** command to start a failover monitor on an OpenVMS node. The failover monitor runs on the OpenVMS node on which the **runmqfm** command is executed.

Syntax

```
►►──runmqfm── -m QMgrName──┬──────┬───────────────────────────────────►◄
                           └─ -d ─┘
```

Required parameters

-m *QMgrName*
Specifies the name of the queue manager for which the **runmqfm** command is to be started. The maximum length supported for the *QMgrName* name is 25 characters.

Optional parameters

-d Specifies that additional debug information is to be logged in the log file.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

The following example starts a failover monitor for a queue manager called `testqm` and writes debug information to a log file called `test.log`.

```
runmqfm -m "testqm" -d > test.log
```

Related commands

failover	Manage a failover set.

runmqlsr (Run listener)

Purpose

The **runmqlsr** (Run listener) command runs a listener process.

Syntax

```
►►──runmqlsr── -t ──┬─ tcp ─┬──────────┬──┬─────────────┬─┬──────────────►
                    │       └─ -p Port─┘  └─ -b Backlog─┘ │
                    └─ lu62 ── -n TpName───────────────────┘

►──┬──────────────┬─────────────────────────────────────────────────────►◄
   └─ -m QMgrName─┘
```

Required parameters

-t Specifies the transmission protocol to be used:

tcp	Transmission Control Protocol / Internet Protocol (TCP/IP)
lu62	SNA LU 6.2. (For the latest information on how to use this parameter, see the release notes in `sys$help:mqseries0510.release_notes`.)

Optional parameters

-p *Port*
Port number for TCP/IP. This flag is valid for TCP and UDP. If a value is not specified, the value is taken from the queue manager configuration file, or from defaults in the program. The default value is 1414.

-n *TpName*
LU 6.2 transaction program name. This flag is valid only for the LU 6.2 transmission protocol. If a value is not specified, the value is taken from the queue manager configuration file. If a value is not given, the command fails.

-m *QMgrName*
Specifies the name of the queue manager. If no name is specified, the command operates on the default queue manager.

-b *Backlog*
Specifies the number of concurrent connection requests that the listener supports. See "The LU62 and TCP stanzas" on page 172 for a list of default values and further information.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

The following command runs a listener on the default queue manager using the TCP/IP protocol. The command specifies that the listener should use port 4321.

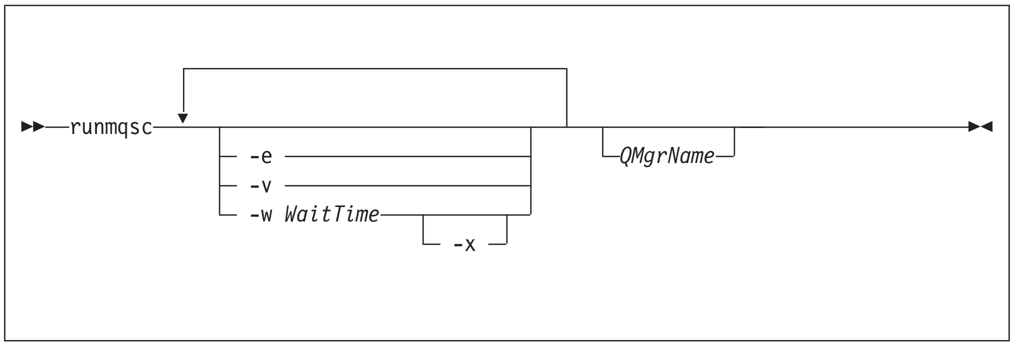
```
runmqlsr -t tcp -p 4321
```

runmqsc (Run MQSeries commands)

Purpose

Use the **runmqsc** command to issue MQSC commands to a queue manager. MQSC commands enable you to perform administration tasks, for example defining, altering, or deleting a local queue object. MQSC commands and their syntax are described in the *MQSeries Application Programming Guide*.

Syntax



Description

You can invoke the **runmqsc** command in three modes:

Verify mode
MQSC commands are verified but not actually run. An output report is generated indicating the success or failure of each command. This mode is only available on a local queue manager.

Direct mode
MQSC commands are sent directly to a local queue manager.

Indirect mode
MQSC commands are run on a remote queue manager. These commands are put on the command queue on a remote queue manager and are run in the order in which they were queued. Reports from the commands are returned to the local queue manager.

Note: The user ID running the remote queue manager needs to exist locally and have the correct authorizations.

The **runmqsc** command takes its input from `SYS$INPUT`. When the commands are processed, the results and a summary are put into a report that is sent to `SYS$OUTPUT`.

By taking `SYS$INPUT` from the keyboard, you can enter MQSC commands interactively.

By redirecting the input from a file you can run a sequence of frequently-used commands contained in the file. You can also redirect the output report to a file.

Optional parameters

-e Prevents source text for the MQSC commands from being copied into a report. This is useful when you enter commands interactively.

-v Specifies verification mode; this verifies the specified commands without performing the actions. This mode is only available locally. The -w and -x flags are ignored if they are specified at the same time.

-w *WaitTime*
: Specifies indirect mode, that is, the MQSC commands are to be run on another queue manager. You must have the required channel and transmission queues set up for this.

WaitTime
: Specifies the time, in seconds, that **runmqsc** waits for replies. Any replies received after this are discarded, however, the MQSC commands are still run. Specify a time between 1 and 999 999 seconds.

  Each command is sent as an Escape PCF to the command queue (SYSTEM.ADMIN.COMMAND.QUEUE) of the target queue manager.

  The replies are received on queue SYSTEM.MQSC.REPLY.QUEUE and the outcome is added to the report. This can be defined as either a local queue or a model queue.

  Indirect mode operation is performed through the default queue manager.

  This flag is ignored if the -v flag is specified.

-x Specifies that the target queue manager is running under MVS/ESA. This flag applies only in indirect mode. The -w flag must also be specified. In indirect mode, the MQSC commands are written in a form suitable for the MQSeries for MVS/ESA command queue.

QMgrName
: Specifies the name of the target queue manager on which the MQSC commands are to be run. If omitted, the MQSC commands run on the default queue manager.

Return codes

0 MQSC command file processed successfully.
10 MQSC command file processed with errors—report contains reasons for failing commands.
20 Error—MQSC command file not run.

Examples

1. Type in this command at the OpenVMS command prompt:

```
runmqsc
```

   Now you can type MQSC commands directly at the OpenVMS command prompt. No queue manager name was specified, therefore, the MQSC commands are processed on the default queue manager.
2. Use this command to specify that MQSC commands are verified only:

```
runmqsc -v BANK < DKA0:[USERS]COMMFILE.IN
```

This verifies the MQSC command file COMMFILE.IN in directory DKA0:[USERS]. The queue manager name is BANK. The output is displayed in the current window.

3. This command runs the MQSC command file MQS_ROOT:[MQM.MQSC]MQSCFILE.IN against the default queue manager.

```
runmqsc < MQS_ROOT:[MQM.MQSC]MQSCFILE.IN > MQS_ROOT:[MQM.MQSC]MQSCFILE.OUT
```

In this example, the output is directed to file MQS_ROOT:[MQM.MQSC]MQSCFILE.OUT.

runmqtmc (Start client trigger monitor)

Purpose

Use the **runmqtmc** command to invoke a trigger monitor for a client. For further information about using trigger monitors, refer to the *MQSeries Application Programming Guide*.

Note: This command is available ***only*** on OpenVMS, OS/2, and AIX clients.

Syntax

```
►►──runmqtmc──┬──────────────┬──┬──────────────────────┬─────────────────────►◄
              └─ -m QMgrName─┘  └─ -q InitiationQName─┘
```

Optional parameters

-m *QMgrName*
: Specifies the name of the queue manager on which the client trigger monitor operates. If omitted, the client trigger monitor operates on the default queue manager.

-q *InitiationQName*
: Specifies the name of the initiation queue to be processed. If omitted, SYSTEM.DEFAULT.INITIATION.QUEUE is used.

Return codes

0	Not used. The client trigger monitor is designed to run continuously and therefore not to end. The value is reserved.
10	Client trigger monitor interrupted by an error.
20	Error—client trigger monitor not run.

runmqtrm (Start trigger monitor)

Purpose

Use the **runmqtrm** command to invoke a trigger monitor. For further information about using trigger monitors, refer to the *MQSeries Application Programming Guide*.

Syntax

```
►►──runmqtrm──┬─────────────────┬──┬──────────────────────┬────────────────►◄
              └─ -m QMgrName ───┘  └─ -q InitiationQName ─┘
```

Optional parameters

-m *QMgrName*
: Specifies the name of the queue manager on which the trigger monitor operates. If omitted, the trigger monitor operates on the default queue manager.

-q *InitiationQName*
: Specifies the name of the initiation queue to be processed. If omitted, SYSTEM.DEFAULT.INITIATION.QUEUE is used.

Return codes

0	Not used. The trigger monitor is designed to run continuously and therefore not to end. The value is reserved.
10	Trigger monitor interrupted by an error.
20	Error— trigger monitor not run.

setmqaut (Set/reset authority)

Purpose

Use the **setmqaut** command to change the authorizations to an object or to a class of objects. Authorizations can be granted to, or revoked from, any number of principals or groups.

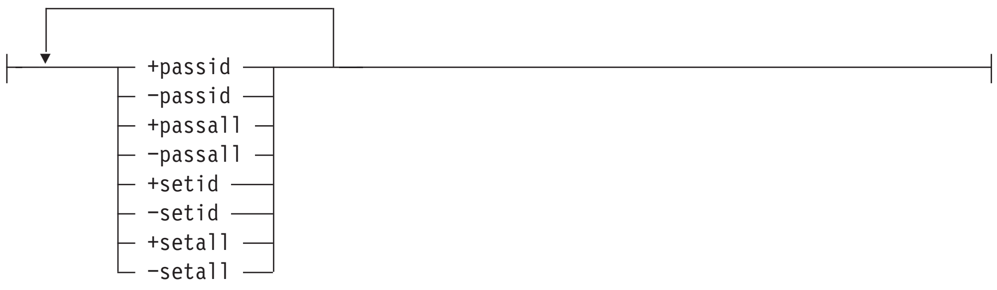
Syntax

```
►►──setmqaut── -m QMgrName── -n ObjectName── -t ObjectType──────────────────►

                                   ┌─────────────────────────┐
                                   ▼                         │
►──┬───────────────────────┬──────┬─ -p PrincipalName─┬──────┴──────────────►
   └─ -s ServiceComponent──┘      └─ -g GroupName─────┘

   ┌──────────────────────────────────────────────┐
   ▼                                              │
►─────┬─┤ MQI authorizations ├──────────────┬─────┴──────────────────────────►◄
      ├─┤ Context authorizations ├──────────┤
      ├─┤ Administration authorizations ├───┤
      └─┤ Generic authorizations ├──────────┘
```

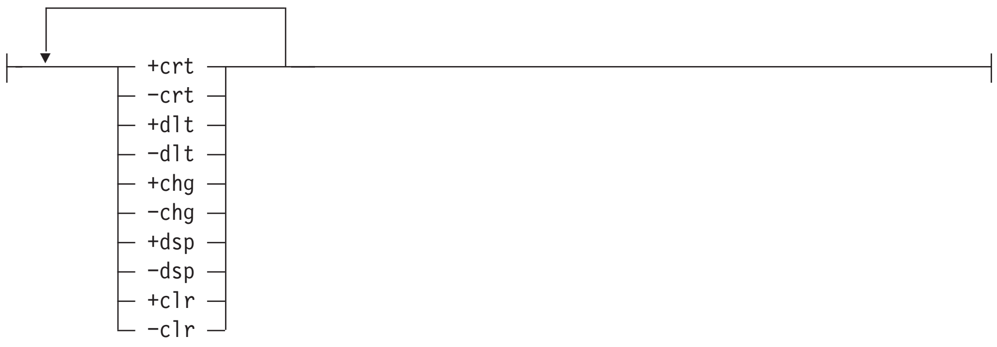
MQI authorizations:

```
    ┌────────────────────┐
    ▼                    │
├────────┬─ +get ──────┬─┴──────────────────────────────────────────────┤
         ├─ −get ──────┤
         ├─ +browse ───┤
         ├─ −browse ───┤
         ├─ +put ──────┤
         ├─ −put ──────┤
         ├─ +inq ──────┤
         ├─ −inq ──────┤
         ├─ +set ──────┤
         ├─ −set ──────┤
         ├─ +connect ──┤
         ├─ −connect ──┤
         ├─ +altusr ───┤
         └─ −altusr ───┘
```

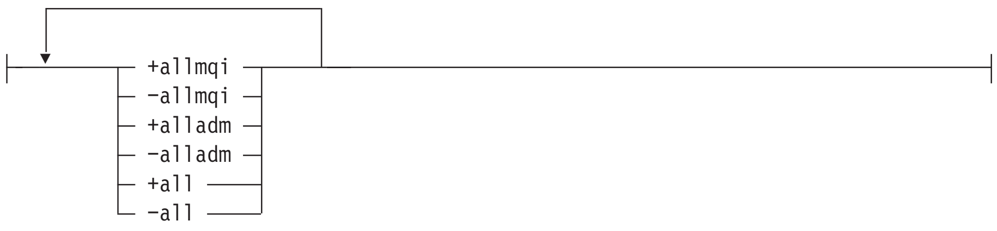
Context authorizations:



Administration authorizations:



Generic authorizations:



Description

You can use this command both to *set* an authorization, that is, give a user group or principal permission to perform an operation, and to *reset* an authorization, that is, remove the permission to perform an operation. You must specify the user groups and principals to which the authorizations apply and also the queue manager, object type, and object name of the object. You can specify any number of groups and principals in a single command.

Attention: If you specify a set of authorizations for a principal, the same authorizations are given to all principals in the same primary group.

The authorizations that can be given are categorized as follows:
- Authorizations for issuing MQI calls
- Authorizations for MQI context
- Authorizations for issuing commands for administration tasks

- Generic authorizations

Each authorization to be changed is specified in an authorization list as part of the command. Each item in the list is a string prefixed by '+' or '–'. For example, if you include `+put` in the authorization list, you are giving authority to issue MQPUT calls against a queue. Alternatively, if you include `–put` in the authorization list, you are removing the authorization to issue MQPUT calls.

Authorizations can be specified in any order provided that they do not clash. For example, specifying `allmqi` with `set` causes a clash.

You can specify as many groups or authorizations as you require in a single command.

If a user ID is a member of more than one group, the authorizations that apply are the union of the authorizations of each group to which that user ID belongs.

Required parameters

-m *`QMgrName`*
: Specifies the name of the queue manager of the object for which the authorizations are to be changed. The name can contain up to 48 characters.

-n *`ObjectName`*
: Specifies the name of the object for which the authorizations are to be changed.

  This is a required parameter ***unless*** it is the queue manager itself. You must specify the name of a queue manager, queue, or process, but must not use a generic name.

-t *`ObjectType`*
: Specifies the type of object for which the authorizations are to be changed.

  Possible values are:
  - **q** or **queue**
  - **prcs** or **process**
  - **qmgr**

Optional parameters

-p *`PrincipalName`*
: Specifies the name of the principal for which the authorizations are to be changed.

  You must have at least one principal or one group.

-g *`GroupName`*
: Specifies the name of the rights identifier representing the user group whose authorizations are to be changed. You can specify more than one rights identifier name, but each name must be prefixed by the -g flag.

-s *`ServiceComponent`*
: This parameter applies only if you are using installable authorization services, otherwise it is ignored.

  If installable authorization services are supported, this parameter specifies the name of the authorization service to which the authorizations apply. This parameter is optional; if it is not specified, the authorization update is made to the first installable component for the service.

`Authorizations`
Specifies the authorizations to be given or removed. Each item in the list is prefixed by a '+' indicating that authority is to be given, or a '–', indicating that authorization is to be removed. For example, to give authority to issue an MQPUT call from the MQI, specify `+put` in the list. To remove authority to issue an MQPUT call, specify `-put.`

Table 17 shows the authorities that can be given to the different object types.

Table 17. Specifying authorizations for different object types

Authority	Queue	Process	Qmgr	Namelist
all	✓	✓	✓	✓
alladm	✓	✓	✓	✓
allmqi	✓	✓	✓	✓
altusr			✓	
browse	✓			
chg	✓	✓	✓	✓
clr	✓			
connect			✓	
crt	✓	✓	✓	✓
dlt	✓	✓	✓	✓
dsp	✓	✓	✓	✓
get	✓			
inq	✓	✓	✓	✓
passall	✓			
passid	✓			
put	✓			
set	✓	✓	✓	
setall	✓		✓	
setid	✓		✓	

Authorizations for MQI calls

altusr Use an alternate user ID in a message.

See the *MQSeries Application Programming Guide* for more information about alternate user IDs.

browse
Retrieve a message from a queue by issuing an MQGET call with the BROWSE option.

connect
Connect the application to the specified queue manager by issuing an MQCONN call.

get Retrieve a message from a queue by issuing an MQGET call.

inq Make an inquiry on a specific queue by issuing an MQINQ call.

put Put a message on a specific queue by issuing an MQPUT call.

set Set attributes on a queue from the MQI by issuing an MQSET call.

Note: If you open a queue for multiple options, you have to be authorized for each of them.

Authorizations for context

passall
Pass all context on the specified queue. All the context fields are copied from the original request.

passid Pass identity context on the specified queue. The identity context is the same as that of the request.

setall Set all context on the specified queue. This is used by special system utilities.

setid Set identity context on the specified queue. This is used by special system utilities.

Authorizations for commands

chg	Change the attributes of the specified object.
clr	Clear the specified queue (PCF Clear queue command only).
crt	Create objects of the specified type.
dlt	Delete the specified object.
dsp	Display the attributes of the specified object.

Authorizations for generic operations

all Use all operations applicable to the object.

alladm
Perform all administration operations applicable to the object.

allmqi Use all MQI calls applicable to the object.

Return codes

0	Command completed normally
36	Invalid arguments supplied
40	Queue manager not available
49	Queue manager stopping
69	Storage not available
71	Unexpected error
72	Queue manager name error
133	Unknown object name
145	Unexpected object name
146	Object name missing
147	Object type missing
148	Invalid object type
149	Entity name missing
150	Authorization specification missing
151	Invalid authorization specification

Examples

1. This example shows a command that specifies that the object on which authorizations are being given is the queue `orange.queue` on queue manager `saturn.queue.manager`.

```
setmqaut -m "saturn.queue.manager" -n "orange.queue" -t queue -g "tango" +inq +alladm
```

The authorizations are being given to user group `tango` and the associated authorization list specifies that user group `tango`:
- Can issue MQINQ calls.
- Has authority to perform all administration operations on that object.

2. In this example, the authorization list specifies that user group `foxy`:
   - Cannot issue any calls from the MQI to the specified queue.
   - Has authority to perform all administration operations on the specified queue.

```
setmqaut -m "saturn.queue.manager" -n "orange.queue" -t queue -g "foxy" -allmqi +alladm
```

3. In this example, the authorization list specifies that user group `waltz` has authority to create and delete queue manager `saturn.queue.manager`:

```
setmqaut -m "saturn.queue.manager" -t qmgr -g "waltz" +crt +dlt
```

Related commands

dspmqaut	Display authority

strmqcsv (Start command server)

Purpose

Use the **strmqcsv** command to start the command server for the specified queue manager. This enables MQSeries to process commands sent to the command queue.

Syntax

```
►►──strmqcsv──┬───────────┬─────────────────────────────────────────────────►◄
              └─QMgrName──┘
```

Optional parameters

QMgrName
: Specifies the name of the queue manager for which the command server is to be started.

Return codes

0	Command completed normally
10	Command completed with unexpected results
20	An error occurred during processing

Examples

The following command starts a command server for queue manager `earth`:

```
strmqcsv "earth"
```

Related commands

endmqcsv	End a command server
dspmqcsv	Display the status of a command server

strmqm (Start queue manager)

Purpose

Use the **strmqm** command to start a local queue manager.

Note: Before using the **strmqm** command, or any other control command, `mqs_startup` must have been run once since the last reboot before the queue manager can be started and run successfully.

Syntax

```
►►──strmqm──┬──────┬──┬──────┬──┬──────────┬──────────────────────────────►◄
            └─ -c ─┘  └─ -z ─┘  └─QMgrName─┘
```

Optional parameters

-c Starts the queue manager, redefines the default and system objects, then stops the queue manager. (The default and system objects for a queue manager are created initially by the **crtmqm** command.) Any existing system and default objects belonging to the queue manager are replaced if you specify this flag.

QMgrName
Specifies the name of a local queue manager to be started. If omitted, the default queue manager is started.

-z Suppresses error messages.

This flag is used within MQSeries to suppress unwanted error messages. Because using this flag could result in loss of information, you should not use it when entering commands on a command line.

Return codes

0	Queue manager started
3	Queue manager being created
5	Queue manager running
16	Queue manager does not exist
23	Log not available
49	Queue manager stopping
69	Storage not available
71	Unexpected error
72	Queue manager name error
100	Log location invalid

Examples

The following command starts the queue manager `account`:

```
strmqm "account"
```

Related commands

crtmqm	Create a queue manager
dltmqm	Delete a queue manager
endmqm	End a queue manager

strmqtrc (Start MQSeries trace)

Purpose

Use the **strmqtrc** command to enable tracing. This command can be run whether tracing is enabled or not. If tracing is already enabled, the trace options in effect are modified to those specified on the latest invocation of the command.

Syntax

```
►►──strmqtrc──┬────────────────┬──┬──────┬──┬─────────────────┬──────────────►
              └─ -m QMgrName ──┘  └─ -e ─┘  └─ -t TraceType ──┘

►──┬──────────────┬───────────────────────────────────────────────────────────►◄
   └─ -l MaxSize ─┘
```

Description

Different levels of trace detail can be requested. For each flow tracetype value you specify, including -t all, specify either -t params or -t detail for any particular trace type, only a default-detail trace is generated for that trace type.

For examples of trace data generated by this command, see “Using MQSeries trace” on page 191.

Optional parameters

-m *QMgrName*
: Is the name of the queue manager to be traced.

  A queue manager name and the -m flag can be specified on the same command as the -e flag. If more than one trace specification applies to a given entity being traced, the actual trace includes all of the specified options.

  It is an error to omit the -m flag and queue manager name, unless the -e flag is specified.

-e
: If this flag is specified, early tracing is requested. Consequently, it is possible to trace the creation or startup of a queue manager. This involves trace information being written, before the processes know to which MQSeries component they belong. Any process, belonging to any component of any queue manager, traces its early processing if this flag is specified. The default, if this flag is not specified, is not to perform early tracing.

-t *TraceType*
: Defines which points during processing can be traced. If this flag is omitted, all trace points are enabled and a full trace generated.

  Alternatively, one or more of the options in the following list can be supplied.

  Note: If multiple trace types are supplied, each ***must*** have its own -t flag. Any number of -t flags can be specified, as long as each has a valid trace type associated with it.

It is not an error to specify the same trace type on multiple -t flags.

all	Output data for every trace point in the system. This is also the default if the -t flag is not specified.
api	Output data for trace points associated with the MQI and major queue manager components.
comms	Output data for trace points associated with data flowing over communications networks.
csflows	Output data for trace points associated with processing flow in common services.
lqmflows	Output data for trace points associated with processing flow in the local queue manager.
remoteflows	Output data for trace points associated with processing flow in the communications component.
otherflows	Output data for trace points associated with processing flow in other components.
csdata	Output data for trace points associated with internal data buffers in common services.
lqmdata	Output data for trace points associated with internal data buffers in the local queue manager.
remotedata	Output data for trace points associated with internal data buffers in the communications component.
otherdata	Output data for trace points associated with internal data buffers in other components.
versiondata	Output data for trace points associated with the version of MQSeries running.
commentary	Output data for trace points associated with comments in the MQSeries components.

-l *MaxSize*

The value of *MaxSize* denotes the maximum size of a trace file (AMQnnnn.TRC) in millions of bytes. For example, if you specify a MaxSize of 1, the size of the trace is limited to 1 million bytes.

When a trace file reaches the specified maximum, it is renamed from AQnnnn.TRC to AMQnnnn.TRS and a new AMQnnnn.TRC file is started. All trace files are restarted when the maximum limit is reached. If a previous copy of an AMQnnnn.TRS file exists, it will be deleted.

Return codes

AMQ7024	This message is issued if arguments that are not valid are supplied to the command.
AMQ8304	The maximum number of nine concurrent traces is already running.

Examples

This command enables tracing of data from common services and the local queue manager, for a queue manager called QM1.

```
strmqtrc -m QM1 -t csdata -t lqmdata
```

Related commands

dspmqtrc	Display formatted trace output
endmqtrc	End MQSeries trace

Part 3. Appendixes

Appendix A. MQSeries for Compaq OpenVMS at a glance

Program and part number

- 5724–A38 MQSeries for Compaq OpenVMS, Alpha Version 5 Release 1, part number 0790997.

Hardware requirements

MQSeries servers can be any Compaq Alpha machine with minimum system disk space of 128 MB.

Software requirements

Software requirements are identical for server and client Compaq OpenVMS environments unless otherwise stated.

Minimum support levels are shown:
- Compaq OpenVMS Alpha Version 7.2–1.

Connectivity

MQSeries for Compaq OpenVMS supports the following network protocols and hardware:

Network protocols:
- SNA LU6.2
- TCP/IP
- DECnet Phase V

And any communications hardware supporting DECnet or TCP/IP, or DIGITAL DECnet/SNA Gateway.

For DECnet connectivity:
- DECnetPLUS Version 7.1 for OpenVMS Version 7.2–1

For TCP/IP connectivity:
- DIGITAL TCP/IP Services for OpenVMS AlphaV5.0.a and V5.1
- Process Software's TCPWare V5.4
- Process Software's Multinet V4.3

For SNA connectivity: SNA APPC LU6.2 software and license must be installed. It must have access to a suitably configured SNA gateway.
- DECnet SNA Gateway ST V1.3
- DECnet SNA LU6.2 API V2.4

Security

MQSeries for Compaq OpenVMS uses the security features of the Object Authority Manager (OAM) for MQSeries for Compaq OpenVMS.

Security

All MQSeries resources run with the VMS Rights Identifier MQM. This rights identifier is created during MQSeries installation and you must grant the rights identifier with this resource attribute to all users who need to control MQSeries resources.

Maintenance functions

MQSeries functions with:

- The **runmqsc** command-line interface.

Compatibility

The MQI for MQSeries for Compaq OpenVMS Alpha, V5.1 is compatible with existing applications running Version 2.2.1.1.

Supported compilers

Programs can be written using C, C++, COBOL, or Java.

- C programs can use the DEC C compiler
- C++ programs can use the DEC C++ compiler
- COBOL programs can use the DEC COBOL compiler
- Java programs can use the Java compiler

Language selection

A supplied message text file is encoded in the 7–bit character set that is native to the OpenVMS operating system.

Internationalization

MQSeries for Compaq OpenVMS lets the CCSID be specified when the queue manager instance is created. The queue manager CCSID defaults to 819. MQSeries for Compaq OpenVMS supports character-set conversion into the configured CCSID of the queue manager. For information about the CCSIDs that can be specified for an MQSeries for Compaq OpenVMS queue manager, including those that provide support for the euro character, see the *MQSeries Application Programming Reference* book.

Appendix B. System defaults

When you create a queue manager using the **crtmqm** control command, the system objects and default objects are created automatically.

- The system objects are those MQSeries objects required for the operation of a queue manager or channel.
- The default objects define all of the attributes of an object. When you create an object, such as a local queue, any attributes that you do not specify explicitly are inherited from the default object.

Table 18. System and default objects for queues

Object Name	**Description**
SYSTEM.DEFAULT.ALIAS.QUEUE	Default alias queue.
SYSTEM.DEFAULT.LOCAL.QUEUE	Default local queue.
SYSTEM.DEFAULT.MODEL.QUEUE	Default model queue.
SYSTEM.DEFAULT.REMOTE.QUEUE	Default remote queue.
SYSTEM.DEAD.LETTER.QUEUE	Sample dead-letter (undelivered-message) queue.
SYSTEM.DEFAULT.INITIATION.QUEUE	Default initiation queue.
SYSTEM.CICS.INITIATION.QUEUE	Default CICS® initiation queue.
SYSTEM.ADMIN.COMMAND.QUEUE	Administration command queue. Used for remote MQSC commands, and PCF commands.
SYSTEM.MQSC.REPLY.QUEUE	MQSC reply-to-queue. This a model queue that creates a temporary dynamic queue for replies to remote MQSC commands.
SYSTEM.ADMIN.QMGR.EVENT	Event queue for queue manager events.
SYSTEM.ADMIN.PERFM.EVENT	Event queue for performance events.
SYSTEM.ADMIN.CHANNEL.EVENT	Event queue for channel events.
SYSTEM.CHANNEL.INITQ	Channel initiation queue.
SYSTEM.CHANNEL.SYNCQ	The queue which holds the synchronization data for channels.
SYSTEM.CLUSTER.COMMAND.QUEUE	The queue used to carry messages to the repository queue manager.
SYSTEM.CLUSTER.REPOSITORY.QUEUE	The queue used to store all repository information.
SYSTEM.CLUSTER.TRANSMIT.QUEUE	The transmission queue for all messages to clusters.

Table 19. System and default objects for channels

Object Name	**Description**
SYSTEM.DEF.SENDER	Default sender channel.
SYSTEM.DEF.SERVER	Default server channel.
SYSTEM.DEF.RECEIVER	Default receiver channel.
SYSTEM.DEF.REQUESTER	Default requester channel.

Table 19. System and default objects for channels (continued)

Object Name	Description
SYSTEM.DEF.SVRCONN	Default server connection channel.
SYSTEM.DEF.CLNTCONN	Default client connection channel.
SYSTEM.AUTO.RECEIVER	Dynamic receiver channel.
SYSTEM.AUTO.SVRCONN	Dynamic server-connection channel.
SYSTEM.DEF.CLUSRCVR	Default receiver channel for the cluster used to supply default values for any attributes not specified when a CLUSRCVR channel is created on a queue manager in a cluster.
SYSTEM.DEF.CLUSSDR	Default sender channel for the cluster used to supply default values for any attributes not specified when CLUSSDR channel is created on a queue manager in the cluster.

Table 20. System and default objects for namelists

Object Name	Description
SYSTEM.DEFAULT.NAMELIST	Default namelist.

Table 21. System and default objects for processes

Object Name	Description
SYSTEM.DEFAULT.PROCESS	Default process definition.

Appendix C. Directory structure

Figure 26 shows the general layout of the data and log directories associated with a specific queue manager. The directories shown apply to the default installation. If you change this, the locations of the files and directories will be modified accordingly.

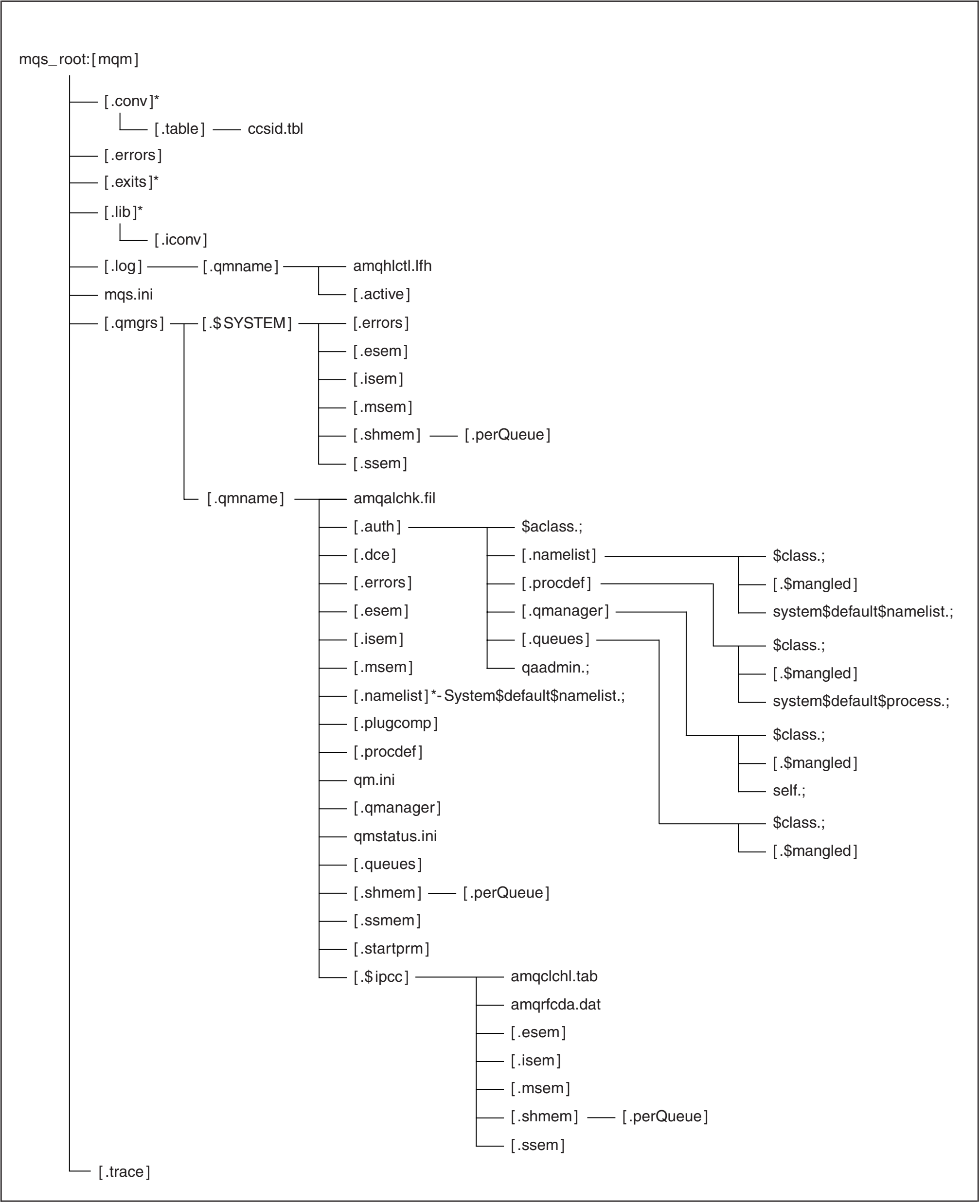


Figure 26. Default directory structure after a queue manager has been started

In Figure 26, the layout is representative of MQSeries after a queue manager has been in use for some time. The actual structure that you have depends on which operations have occurred on the queue manager.

Directories and files in MQS_ROOT:[MQM]

By default, the following directories and files are located in the directory MQS_ROOT:[MQM]:

.conv This directory contains all files used for data conversion.

.table This directory contains the ccsid.tbl. file.

.errors This directory contains the operator message files, from newest to oldest:
AMQERR01.LOG
AMQERR02.LOG
AMQERR03.LOG

.exits An empty directory to contain user-written exits.

.lib This directory contains the subdirectory .iconv. The subdirectory contains all the codeset conversion tables.

.iconv A directory containing codeset conversion tables (such as `002501B5.TBL` to `44B031A8.TBL`).

.log This directory contains the following subdirectory and files after you have installed MQSeries, created and started a queue manager, and have been using that queue manager for some time.

amqhlctl.lfh
Log control file.

active This directory contains the log files, numbered as follows:
S0000000.LOG
S0000001.LOG
S0000002.LOG
... and so on.

mqs.ini
MQSeries configuration file.

.qmgrs
This directory contains a subdirectory `.$system` and a subdirectory `.qmname` for each queue manager. The `.$system` directory contains directories and files used internally by MQSeries. For more information about the `.qmname` subdirectory, see "Directories and files in the MQS_ROOT:[MQM.QMGRS.QMNAME] subdirectory".

.trace This directory contains the trace files created from the **strmqtrc** command.

Directories and files in the MQS_ROOT:[MQM.QMGRS.QMNAME] subdirectory

By default, the following directories and files are located in the directory MQS_ROOT:[MQM.QMGRS.QMNAME]. The .QMNAME is created for every queue manager created and running on the system.

amqalchk.fil
Checkpoint file containing information about last checkpoint.

.auth This directory contains subdirectories and files associated with authority.

$aclass.;
This file contains the authority stanzas for all classes.

.namelist
This directory contains a file for each namelist. Each file contains the authority stanzas for the associated namelist.

$class.;
This file contains the authority stanzas for the namelist class.

.$mangled
When namelist names contain invalid OpenVMS characters, they are automatically converted to valid OpenVMS names. The valid OpenVMS names are held in this file. See “Understanding MQSeries file names” on page 19.

system$default$namelist
This file contains authority stanzas for the system default namelist.

.procdef
Each MQSeries process definition is associated with a file in this directory.

$class.;
This file contains the authority stanzas for the process definition class.

.$mangled
When process definition names contain invalid OpenVMS characters, they are automatically converted to valid OpenVMS names. The valid OpenVMS names are held in this file. See “Understanding MQSeries file names” on page 19.

.system$default$process.;
This file contains authority stanzas for the system default processes.

.qmanager
This directory contains a file for each queue manager. Each file contains the authority stanzas for the associated queue manager.

$class.;
This file contains the authority stanzas for the queue manager class.

.$mangled
When queue manager definition names contain invalid OpenVMS characters, they are automatically converted to valid OpenVMS names. The valid OpenVMS names are held in this file. See “Understanding MQSeries file names” on page 19.

self.; This file contains the authority stanzas for the queue manager object.

.queues
This directory contains a file for each queue. Each file contains the authority stanzas for the associated queue.

$CLASS
This file contains the authority stanzas for the queue class.

.$mangled
When queue names contain invalid OpenVMS characters, they are automatically converted to valid OpenVMS names. The valid OpenVMS names are held in this file. See "Understanding MQSeries file names" on page 19.

Definition files for the queue
Each file corresponds to an object predefined for the queue manager.
system$admin$channel$event.;
system$admin$command$queue.;
system$admin$perfm$event.;
system$admin$qmgr$event.;
system$channel$initq.;
system$channel$syncq.;
system$cics$initiation$queue.;
system$cluster$command$queue.;
system$cluster$repository$queue.;
system$cluster$transmit$queue.;
system$dead$letter$queue.;
system$default$alias$queue.;
system$default$initiation$queue.;
system$default$local$queue.;
system$default$model$queue.;
system$default$remote$queue.;
system$mqsc$reply$queue.;

.qaadmin.;
File used internally for controlling authorizations.

.dce Empty directory reserved for use by DCE support.

.errors This directory contains the operator message files, from newest to oldest:
amqerr01.log
amqerr02.log
amqerr03.log

.esem Directory containing files used internally.

.isem Directory containing files used internally.

.msem Directory containing files used internally.

.namelist
This directory contains namelists for each queue manager.

.plugcomp
This empty directory is reserved for use by installable services.

.procdef
Each MQSeries process definition is associated with a file in this directory. The filename matches the process definition name.

qm.ini Queue manager configuration file.

.qmanager
The queue manager object.

qmstatus.ini
This file contains text describing the status of the queue manager.

.queues
Each queue has a directory in here containing a single file called 'q'.

The file name matches the queue name—subject to certain restrictions; see "Understanding MQSeries file names" on page 19.

.shmem

- **.perQueue**
  Directory containing files used internally.

.ssem Directory containing files used internally.

.startprm
Directory containing temporary files used internally.

.$ipcc

- **amqclchl.tab**
  Client channel table file.
- **amqrfcda.dat**
  Channel table file.
- **.esem** Directory containing files used internally.
- **.isem** Directory containing files used internally.
- **.msem** Directory containing files used internally.
- **.shmem**
  - **.perQueue**
    Directory containing files used internally.
- **.ssem** Directory containing files used internally.

Directory structure

Appendix D. Comparing command sets

The following tables compare the facilities available from the different administration command sets:

- "Commands for queue manager administration"
- "Commands for command server administration"
- "Commands for queue administration" on page 306
- "Commands for process administration" on page 306
- "Commands for channel administration" on page 307
- "Other control commands" on page 307

Note: Only MQSC commands that apply to MQSeries for Compaq OpenVMS are shown.

Commands for queue manager administration

Table 22. Commands for queue manager administration

PCF	MQSC	Control
Change Queue Manager	ALTER QMGR	–
(Create queue manager)★	–	crtmqm
(Delete queue manager)★	–	dltmqm
Inquire Queue Manager	DISPLAY QMGR	–
(Stop queue manager)★	–	endmqm
Ping Queue Manager	PING QMGR	–
(Start queue manager)★	–	strmqm
Note: ★ Not available as PCF commands.		

Commands for command server administration

Table 23. Commands for command server administration

Description	Control
Display command server	dspmqcsv
Start command server	strmqcsv
Stop command server	endmqcsv
Note: Functions in this group are available only as control commands. There are no equivalent MQSC or PCF commands in this group.	

Commands for queue administration

Table 24. Commands for queue administration

PCF	MQSC
Change Queue	ALTER QLOCAL ALTER QALIAS ALTER QMODEL ALTER QREMOTE
Clear Queue	CLEAR QUEUE
Copy Queue	DEFINE QLOCAL(x) LIKE(y) DEFINE QALIAS(x) LIKE(y) DEFINE QMODEL(x) LIKE(y) DEFINE QREMOTE(x) LIKE(y)
Create Queue	DEFINE QLOCAL DEFINE QALIAS DEFINE QMODEL DEFINE QREMOTE
Delete Queue	DELETE QLOCAL DELETE QALIAS DELETE QMODEL DELETE QREMOTE
Inquire Queue	DISPLAY QUEUE
Inquire Queue Names	DISPLAY QUEUE
Note: There are no equivalent control commands in this group.	

Commands for process administration

Table 25. Commands for process administration

PCF	MQSC
Change Process	ALTER PROCESS
Copy Process	DEFINE PROCESS(x) LIKE(y)
Create Process	DEFINE PROCESS
Delete Process	DELETE PROCESS
Inquire Process	DISPLAY PROCESS
Inquire Process Names	DISPLAY PROCESS
Note: There are no equivalent control commands in this group.	

Commands for channel administration

Table 26. Commands for channel administration

PCF	MQSC	Control
Change Channel	ALTER CHANNEL	–
Copy Channel	DEFINE CHANNEL(x) LIKE(y)	–
Create Channel	DEFINE CHANNEL	–
Delete Channel	DELETE CHANNEL	–
Inquire Channel	DISPLAY CHANNEL	–
Inquire Channel Names	DISPLAY CHANNEL	–
Ping Channel	PING CHANNEL	–
Reset Channel	RESET CHANNEL	–
Resolve Channel	RESOLVE CHANNEL	–
Start Channel	START CHANNEL	runmqchl
Start Channel Initiator	START CHINIT	runmqchi
Start Channel Listener	–	runmqlsr
Stop Channel	STOP CHANNEL	–

Other control commands

Table 27. Other control commands

Description	Control
Create MQSeries conversion exit	crtmqcvx
Display authority	dspmqaut
Display files used by objects	dspmqfls
Display MQSeries formatted trace output	dspmqtrc
End MQSeries trace	endmqtrc
Manage a failover set	failover
Record media image	rcdmqimg
Recreate media object	rcrmqobj
Resolve MQSeries transactions	rsvmqtrn
Run MQSC commands	runmqsc
Run trigger monitor	runmqtrm
Run client trigger monitor	runmqtmc
Set or reset authority	setmqaut
Start a failover monitor	runmqfm
Start MQSeries trace	strmqtrc
Note: Functions in this group are available only as control commands. There are no direct PCF or MQSC equivalents.	

Appendix E. Sample MQI programs and MQSC files

MQSeries for Compaq OpenVMS provides a set of short sample MQI programs and MQSC command files that you can use and experiment with. These are described in the following sections:

- "MQSC command file samples"
- "C and COBOL program samples"
- "Miscellaneous tools" on page 310

MQSC command file samples

Table 28 lists the MQSC command file samples. These are simply ASCII text files containing MQSC commands. You can invoke the **runmqsc** command against each file in turn to create the objects specified in the file. See "Running the supplied MQSC command files" on page 38.

By default, these files are located in directory `MQS_EXAMPLES:`

Table 28. MQSC command files

File name	Purpose
AMQSCOS0.TST	Creates a set of MQI objects for use with the C and COBOL program samples.

C and COBOL program samples

Table 29 lists the sample MQI source files. By default, the source files are located in directory `MQS_EXAMPLES:` and the compiled versions in `[.BIN]` directory under `MQS_EXAMPLES:`. To find out more about what the programs do and how to use them, see the *MQSeries Application Programming Guide*.

Table 29. Sample programs - source files

C	COBOL	Purpose
AMQSBCG0.C	–	Reads and then outputs both the message descriptor and message context fields of all the messages on a specified queue.
AMQSECHA.C	AMQVECHX.COB	Echoes a message from a message queue to the reply-to queue. Can be run as a triggered application program.
AMQSGBR0.C	AMQ0GBR0.COB	Writes messages from a queue to SYS$OUTPUT leaving the messages on the queue. Uses MQGET with the browse option.
AMQSGET0.C	AMQ0GET0.COB	Removes the messages from the named queue (using MQGET) and writes them to SYS$OUTPUT.
AMQSINQA.C	AMQVINQX.COB	Reads the triggered queue; each request read as a queue name; responds with information about that queue.
AMQSPUT0.C	AMQ0PUT0.COB	Copies SYS$INPUT to a message and then puts this message on a specified queue.
AMQSREQ0.C	AMQ0REQ0.COB	Puts request messages on a specified queue and then displays the reply messages.
AMQSSETA.C	AMQVSETX.COB	Inhibits puts on a named queue and responds with a statement of the result. Runs as a triggered application.

Table 29. Sample programs - source files (continued)

C	COBOL	Purpose
AMQSTRG0.C	–	A trigger monitor that reads a named initiation queue and then starts the program associated with each trigger message. Provides a subset of the full triggering function of the supplied **runmqtrm** command.
AMQSVFCX.C	–	A sample C skeleton of a Data Conversion exit routine.

Miscellaneous tools

These tool files are provided to support the formatter and code conversion.

Table 30. Miscellaneous files

File name	Location	Purpose
AMQTRC.FMT	SYS$LIBRARY	Defines MQSeries trace formats.
CCSID.TBL	MQS_ROOT:[MQM.CONV.TABLE]	Edit this file to add any newly supported CSSID values to your MQSeries system. For more information about CCSID, see the CDRA (Character Data Representation Architecture) documentation.

Appendix F. OpenVMS cluster failover set templates

This appendix contains the following failover set templates:

- “Template Configuration File FAILOVER.TEMPLATE”
- “Template StartCommand procedure START_QM.TEMPLATE” on page 313
- “Template EndCommand procedure END_QM.TEMPLATE” on page 314
- “Template TidyCommand procedure TIDY_QM.TEMPLATE” on page 317

Template Configuration File FAILOVER.TEMPLATE

```
#******************************************************************#
#*                                                                *#
#* Statement:     Licensed Materials - Property of IBM            *#
#*                                                                *#
#*                33H2205, 5622-908                               *#
#*                33H2267, 5765-623                               *#
#*                29H0990, 5697-176                               *#
#*                (C) Copyright IBM Corp. 2000, 2001              *#
#*                                                                *#
#******************************************************************#
#
# FAILOVER.TEMPLATE
# Template for creating a FAILOVER.INI configuration file
# All lines beginning with a '#' are treated as comments
#
# OpenVMS Cluster Failover Set Configuration information
# -------------------------------------------------------
#
# The TCP/IP address used by the OpenVMS Cluster Failover Set
#
IpAddress=n.n.n.n
#
# The TCP/IP port number used by the MQSeries Queue Manager
#
PortNumber=1414
#
# The timeout used by the EndCommand command procedure
#
TimeOut=30
#
# The command procedure used to start the Queue Manager
#
StartCommand=@sys$manager:start_qm
#
# The command procedure used to end the Queue Manager
#
EndCommand=@sys$manager:end_qm
#
# The command procedure used to tidy up on a node after a
# Queue Manager failure but the OpenVMS node did not fail
#
TidyCommand=@sys$manager:tidy_qm
#
# The directory in which the log files for the start, end and
# tidy commands are written
#
LogDirectory=mqs_root:[mqm.errors]
#
# The number of nodes in the OpenVMS Cluster Failover Set. The
# number of nodes defined below must agree with this number
```

FAILOVER.TEMPLATE

```
#
NodeCount=2
#
# The Name of the OpenVMS node
#
NodeName=BATMAN
#
# The TCP/IP interface name for the node
#
Interface=we0
#
# The priority of the node
#
Priority=1
#
# The Name of the OpenVMS node
#
NodeName=ROBIN
#
# The TCP/IP interface name for the node
#
Interface=we0
#
# The priority of the node
#
Priority=2
```

Figure 27. Template configuration file: failover.template

Template StartCommand procedure START_QM.TEMPLATE

```
$ on error then exit
$!*****************************************************************
$!* Statement:     Licensed Materials - Property of IBM            *
$!*                33H2205, 5622-908                               *
$!*                33H2267, 5765-623                               *
$!*                29H0990, 5697-176                               *
$!*                (C) Copyright IBM Corp. 2000, 2001              *
$!*****************************************************************
$! Template command procedure used by Failover Sets to start the
$! queue manager
$! Parameters :
$! P1 = Queue Manager Name
$! P2 = Queue Manager Directory Name
$! P3 = TCP/IP address
$! P4 = TCP/IP interface name
$! P5 = Listener port number
$!
$ @sys$startup:mqs_symbols
$ set def mqs_root:[mqm.qmgrs.'p2'.errors]
$ define sys$scratch mqs_root:[mqm.qmgrs.'p2'.errors]
$!
$! Digital TCP/IP Services for OpenVMS commands
$!
$ @sys$startup:tcpip$define_commands
$!
$! Configure the IP address
$!
$ ifconfig 'p4' alias 'p3'
$!
$! TCPware for OpenVMS commands
$!
$! @tcpware:tcpware_commands
$!
$! Configure the IP address
$!
$! netcu add secondary 'p3'
$!
$! MultiNet for OpenVMS commands
$!
$! Configure the IP address
$!
$! define/sys/exec multinet_ip_cluster_aliases "''p3'"
$!
$! Restart the Multinet server
$!
$! @multinet:start_server
$!$! Start the queue manager
$!
$ strmqm 'p1'
$!
$! Start the listener
$!
$! runmqlsr -t tcp -p 'p5' -m 'p1'
$!
$! Insert commands to start any applications
$!
$exit
```

Figure 28. Template StartCommand procedure: Start_QM.template

Template EndCommand procedure END_QM.TEMPLATE

```
$ on error then exit
$!
$!*****************************************************************
$!*                                                               *
$!* Statement:     Licensed Materials - Property of IBM           *
$!*                                                               *
$!*                33H2205, 5622-908                              *
$!*                33H2267, 5765-623                              *
$!*                29H0990, 5697-176                              *
$!*                (C) Copyright IBM Corp. 2000, 2001             *
$!*                                                               *
$!*****************************************************************
$!
$! Template Command procedure used by Failover Sets to end the
$! queue manager
$!
$! Parameters :
$!
$! P1 = Queue Manager Name
$! P2 = Queue Manager Directory Name
$! P3 = TCP/IP address
$! P4 = TCP/IP interface name
$! P5 = Listener port number
$! P6 = End Queue Manager Timeout
$!
$ @sys$startup:mqs_symbols
$ check_qm:==$sys$system:mqcheckqm
$ set def mqs_root:[mqm.qmgrs.'p2'.errors]
$ define sys$scratch mqs_root:[mqm.qmgrs.'p2'.errors]
$ SS$_NORMAL=1
$ SS$_ABORT=44
$ SS$_TIMEOUT=556
$!
$! Insert commands to shutdown any applications prior to ending MQSeries
$!
$! Get the timeout period for each operation seconds
$!
$ timeout = 'p6'
$!
$! Initialise the outer loop
$!
$ out_count = 0
$!
$! Initialise the complete flag
$!
$ complete  = 0
$!
$ out_next:
$ if (out_count .gt. 2) .or. (complete .eq. 1) then goto out_finish
$!
$ if out_count .eq. 0
$ then
$!
$! End the queue manager gracefully first
$!
$ spawn/nowait $endmqm -i 'p1'
$ else
$ if out_count .eq. 1
$ then
$!
$! End the queue manager abruptly
$!
$ spawn/nowait $endmqm -p 'p1'
$ else
$!
```

```
$! Stop/id the execution controller
$!
$ check_qm -m 'p1'
$ if ( mqs$ec_pid .nes. "") then $stop/id='mqs$ec_pid'
$ endif
$ endif
$!
$ in_start:
$!
$! Initialise the outer loop
$!
$ in_count  = 0
$!
$ in_next:
$!
$! Inner loop
$!
$ if ( ( in_count .ge. timeout) .and. ( timeout .ne. 0 ) ) -
     .or. (complete .eq. 1) then goto in_finish
$!
$! Check if the execution controller is still running
$!
$ check_qm -m 'p1'
$ if mqs$ec_pid .eqs. ""
$ then
$!
$! The Execution controller is no longer running so we are finished
$!
$ complete = 1
$ goto in_finish
$ endif
$!
$! Wait a second and go round again
$!
$ wait 00:00:01
$ in_count = in_count + 1
$ goto in_next
$ in_finish:
$!
$! End of the inner loop
$!
$ out_count = out_count + 1
$ goto out_next
$ out_finish:
$!
$! End of the outer loop
$!
$! Digital TCP/IP Services for OpenVMS commands
$!
$ @sys$startup:tcpip$define_commands
$!
$! De-configure the IP address
$!
$ ifconfig 'p4' -alias 'p3'
$!
$! TCPware for OpenVMS commands
$!
$! @tcpware:tcpware_commands
$!
$! De-configure the IP address
$!
$! netcu remove secondary 'p3'
$!
$! MultiNet for OpenVMS commands
$!
$! De-configure the IP address
$!
```

```
$! deass/sys/exec multinet_ip_cluster_aliases
$!
$! Restart the Multinet server
$!
$! @multinet:start_server
$!
$!
$! If the Queue Manager was shutdown successfully set the status
$! to SS$_NORMAL. If it was necessary to STOP/ID the Execution
$! controller set the status to SS$_ABORT and if the Execution
$! controller is still running set the status to SS$_TIMEOUT to
$! indicate an error
$!
$ if ( complete .eq. 1 )
$then
$!
$! End the listener process
$!
$!  endmqlsr -m 'p1'
$!
$ if ( out_count .eq. 3 )
$ then
$ exit SS$_ABORT
$ else
$ exit SS$_NORMAL
$ endif
$else
$ exit SS$_TIMEOUT
$endif
```

Figure 29. Template EndCommand procedure: END_QM.template

Template TidyCommand procedure TIDY_QM.TEMPLATE

```
$ on error then exit
$!*******************************************************************
$!* Statement:     Licensed Materials - Property of IBM             *
$!*                                                                 *
$!*                33H2205, 5622-908                                *
$!*                33H2267, 5765-623                                *
$!*                29H0990, 5697-176                                *
$!*                (C) Copyright IBM Corp. 2000, 2001               *
$!*******************************************************************
$! Template Command procedure used by Failover Sets to tidy up after
$! a queue manager failure
$!
$! Parameters :
$! P1 = Queue Manager Name
$! P2 = Queue Manager Directory Name
$! P3 = TCP/IP address
$! P4 = TCP/IP interface name
$! P5 = Listener port number
$!
$ @sys$startup:mqs_symbols
$ set def mqs_root:[mqm.qmgrs.'p2'.errors]
$ define sys$scratch mqs_root:[mqm.qmgrs.'p2'.errors]
$!
$! Insert commands to do any tidying up after a queue manager has failed
$!
$! Digital TCP/IP Services for OpenVMS commands
$!
$ @sys$startup:tcpip$define_commands
$!
$! De-configure the IP address
$!
$ ifconfig 'p4' -alias 'p3'
$!
$! TCPware for OpenVMS commands
$!
$! @tcpware:tcpware_commands
$!
$! De-configure the IP address
$!
$! netcu remove secondary 'p3'
$!
$! MultiNet for OpenVMS commands
$!
$! De-configure the IP address
$!
$! deass/sys/exec multinet_ip_cluster_aliases
$!
$! Restart the Multinet server
$!
$! @multinet:start_server
$!
$exit
```

Figure 30. Template TidyCommand procedure: TIDY_QM.template

Appendix G. Codeset support on MQSeries for Compaq OpenVMS

MQSeries for Compaq OpenVMS supports most of the codesets used by the locales – that is, the subsets of the user's environment which define the conventions for a specific culture – that are provided as standard on MQSeries for Compaq OpenVMS.

If the locale is not set the CCSID used is 819 - the ISO8859-1 codeset.

The CCSID (Coded Character Set Identifier) used in MQSeries to identify the codeset used for the message and message header data is obtained by analyzing the LC_CTYPE category of the locale configuration.

Table 31 shows the locales and the CCSIDs that are registered for the codeset used by the locale.

Table 31. Locales and CCSIDs

Locale	Language	codeset	CCSID
C	English	ISO8859-1	819
CS_CZ_ISO8859-2	Czech	ISO8859-2	912
DA_DK_ISO8859-1	Danish	ISO8859-1	819
DE_DE_ISO8859-1	German	ISO8859-1	819
DE_CH_ISO8859-1	German - Switzerland	ISO8859-1	819
EL_GR_ISO8859-7	Greek	ISO8859-7	813
EN_GB_ISO8859-1	English - United Kingdom	ISO8859-1	819
EN_US_ISO8859-1	English - USA	ISO8859-1	819
ES_ES_ISO8859-1	Spanish	ISO8859-1	819
FI_FI_ISO8859-1	Finnish	ISO8859-1	819
FR_FR_ISO8859-1	French - France	ISO8859-1	819
FR_BE_ISO8859-1	French - Belgium	ISO8859-1	819
FR_CA_ISO8859-1	French - Canada	ISO8859-1	819
FR_CH_ISO8859-1	French - Switzerland	ISO8859-1	819
HU_HU_ISO8859-2	Hungarian	ISO8859-2	912
IS_IS_ISO8859-1	Icelandic	ISO8859-1	819
IT_IT_ISO8859-1	Italian - Italy	ISO8859-1	819
IW_IL_ISO8859-8	Hebrew	ISO8859-8	916
JA_JP_EUCJP	Japanese	eucJP	954
JA_JP_SDECKANJI	Japanese	SDECKANJI	954**
JA_JP_SJIS	Japanese	SJIS	932
KO_KR_DECKOREAN	Korean	DECKOREAN	970**
NL_NL_ISO8859-1	Dutch - Netherlands	ISO8859-1	819

Table 31. Locales and CCSIDs (continued)

Locale	Language	codeset	CCSID
NL_BE_ISO8859-1	Dutch - Belgium	ISO8859-1	819
NO_NO_ISO8859-1	Norwegian	ISO8859-1	819
PL_PL_ISO8859-2	Polish	ISO8859-2	912
PT_PT_ISO8859-1	Portuguese	ISO8859-1	819
SK_SK_ISO8859-2	Slovak	ISO8859-2	912
RU_RU_ISO8859-5	Cyrillic	ISO8859-5	915
SV_SE_ISO8859-1	Swedish	ISO8859-1	819
TR_TR_ISO8859-9	Turkish	ISO8859-9	920
ZH_CN_DECHANZI	Chinese - Simplified	DECHANZI	1383**
ZH_HK_DECHANZI	Chinese - Simplified	DECHANZI	1383**
ZH_HK_EUCTW	Chinese - Traditional	eucTW	964
ZH_HK_EUCTW	Chinese - Traditional	eucTW	964
ZH_HK_DECHANYU	Chinese - Traditional	DECHANYU	964**
ZH_TW_DECHANYU	Chinese - Traditional	DECHANYU	964**
ZH_HK_BIG5	Chinese - Traditional	big5	950
ZH_TW_BIG5	Chinese - Traditional	big5	950

Note:

** The CCSID used is the nearest registered IBM CCSID.

For further information listing inter-platform support for these locales, see the *MQSeries Application Programming Reference* book.

Appendix H. MONMQ diagnostic utility

The MONMQ utility is a tool to assist in the diagnosis and resolution of problems with MQSeries for Compaq OpenVMS. The MONMQ utility can be used interactively, from the command line, or from within a DCL script.

The MONMQ utility is most commonly used to:
- Manage shared memory
- Help gather OpenVMS resource usage information
- Obtain trace output from a running queue manager.

MONMQ has a help system to assist with parameters and can also run a script of MONMQ commands. When MONMQ starts, a default script `sys$manager:mqs_trace_startup.mqt` is run to provide an initial configuration.

```
$monmq
ok - trace mailbox 0 opened as default

MQT> help
Help can be used to display information about available commands or parameters
Help [ <verb> || <parameter/variable name> || commands || parameters || examples]

Valid trace commands are in the format:
Verb [<parameters>] [<variable = expression>][;][optional second command]
```

Overview

Tracing MQSeries on OpenVMS is implemented using global sections and mailboxes. Up to ten trace sections (LUs) can coexist on any one node where MQSeries is installed. However, it is strongly recommended that a trace session only ever employs one LU at any time. It is also not advisable for more than one user to have the same LU open at any one time. The results of either of these conditions are unpredictable.

Each shared section (LU) contains the channel definitions and the LU definition itself. Each channel definition contains the connected thread details, the threads private stack and the threads circular buffer. Furthermore the shared section contains a set of flags used for interprocess communication between MONMQ and the connected threads.

For each LU there is an associated mailbox used for receiving realtime trace messages. To perform realtime tracing, a client process must be initiated using the **TRACE START** command. This dedicated detached process reads, formats and displays each message as it arrives in the LUs mailbox. Each connected thread writes to the same mailbox and thus provides you with the ability to physically view the intercommunication between MQSeries processes/threads.

MONMQ, if driven correctly, can provide a comprehensive method for diagnosing problems such as, interprocess timing problems, exhausted operating system resources or even coding problems.

The MONMQ commands are described in this appendix.

Variables within MONMQ

Many commands within MONMQ make use of variables. A variable uses a default value, defined by the **set** command, if one is not specified within the command. When a variable is used with a command other than set, the default value for that variable is not changed.

Variables can contain:

- Integer variables (either decimal or hexadecimal).

  Hexadecimal values can be entered with a leading 0x, or by entering a value with letters a-f where a hexadecimal value is expected.
- Text, which must be quoted.
- A range, which is entered by putting minimum:maximum.

  A range is used so that, for example, a command can apply to a range of channels.

For example:

```
MQT> set lu=2
MQT> set pid=0x223
MQT> set pid=2fa
MQT> set buffile="filename.buf"
MQT> set chl=0:20
```

The current default value for the variables can be displayed by using the variables command.

```
MQT> variables
defined variables
lu=0:0                     nochls=20                  buffer=1000
chl=0:20                   component=0(HEX)           line=0
mask=0(HEX)                pid=0(HEX)                 node=(null)
function=0(HEX)            div=0                      depth=32
resource=0                 wait=1(BOOL)               timestamp=0(BOOL)
listfile=(null)            buffile=(null)             step=0(BOOL)
active=0(BOOL)             fname=(null)               delay=100
post=0(BOOL)


defined constants
fent=1(HEX)                fout=2(HEX)                ferr=4(HEX)
fxxx=8(HEX)                dgn=10(HEX)                shm=20(HEX)
spl=40(HEX)                evt=80(HEX)                mtx=100(HEX)
prc=200(HEX)               msc=400(HEX)               inf=800(HEX)
log=2000(HEX)              shl=4000(HEX)              memory=3
mutex=4                    mailbox=5                  nanoseconds=1
microseconds=2             milliseconds=3             seconds=4
```

You are recommended to set default values to simplify the commands. For example, the following command sequences are functionally identical:

```
MQT> open lu=0 buffer=1000 nochls=20
MQT> open lu=1 buffer=1000 nochls=20
MQT> show channels lu=0 chl=1:10
MQT> show channels lu=1 chl=1:10
```

or

```
MQT> set nochls=20 chl=0:10 buffer=1000 lu=0
MQT> open
MQT> open lu=1
MQT> show channels
MQT> show channels lu=1
```

MONMQ commands can be abbreviated to the minimum number of characters required to ensure a unique command. This series of commands could be shortened still further to:

```
MQT> se noc=20 ch=0:10 buffe=1000 lu=0
MQT> op
MQT> op lu=1
MQT> sh ch
MQT> sh ch lu=1
```

MONMQ can also perform simple arithmetic operations with variables so that commands such as set lu=lu+1 is possible.

New variables can be declared with the **declare** command. The parameters are:
- Variable name
- Variable type
- Help text. The help command can then retrieve the help text.

```
MQT> declare ec int "channel number for execution controller"
MQT> set ec = 4
MQT> show channel chl=ec
Chl    Pid    Mailbox    Stack   Active  Post    Time   Mask     Process Name
  4    2c1f   7ee70290     4        0      0      0    ffffffff   AMQZXMA0.EXE
MQT> help ec

VARIABLE ec:
channel number for execution controller

MQT>
```

Assigning default values

DEFAULT variable=<*expression*> [variable=*expression*] ...

This command allows default values to be assigned to all variables defined within MONMQ. Once set, the default variable name can then be omitted from the command line. For example:

```
MQT>default lu=2 chl=3:6
```

This command sets the default value 2 to the lu variable and the values 3 to 6 inclusive to the channel variable. From now on, when using a typical command such as **show channels**, channels 3 to 6 inclusive on lu 2 is displayed.

Default values are used only where the variable is omitted from the command line. All default values are set in the startup script file `MQS_TRACE_STARTUP.MQT`. This file can be edited to suit your needs.

Opening or creating a trace section and associated mailbox

OPEN [lu=*number*] [nochls=*number*] [buffer=*number*]

This command opens or creates a trace section and associated mailbox. The **open** command creates the basic resource required for tracing MQSeries processes. Each LU has an associated shared section and mailbox used to communicate with MQSeries processes.

This command takes three optional parameters. The first parameter [LU] is the number assigned to the trace section/mailbox and is used as a reference by most other MONMQ commands. A maximum of ten LUs may be created on a single node. The default value is zero. If the specified LU already exists then MONMQ connects to the existing trace section. If no section exists then a new section is created.

The second parameter [nochls] specifies the number of channels that this LU will have. Each channel represents a single MQSeries process/thread connection. The default value is 20.

The third parameter [buffer] specifies the maximum size of the trace history buffer for each channel. The default is 1000.

You must have at least one LU open before being able to perform other MONMQ commands.

Displaying the logical unit definition

SHOW SEGMENT [lu=*range*]

This command displays the Logical Unit definition. An example of the output is shown below with a brief description along side each field when you type the command **show segment lu=0**.

```
Trace LU          : 0             /* The LU number as specified in the OPEN [lu] parameter.
Mailbox name      : MQS_TRC_MBX_0 /* The permanent mailbox name assigned to this LU
Device name       : MBA1065:      /* The device name of the mailbox
Status            : Disabled      /* The current status of the mailbox ie.
Mailbox channel   : 352           /* The mailbox channel number assigned to the MONMQ process
History buffer size: 1000         /* The maximum number of message entries in the history
                                  /* circular buffer (as specified by the OPEN [buffer] parameter)
Threads mapped #  : 1             /* The number of processes/threads mapped to this LUs global
                                   /* section (MONMQ always attached)
Time stamping     : Enabled       /* Global timestamp flag (not yet implemented)
Max channels #    : 20            /* Number of channels defined for this LU as specified by the
                                  /* OPEN [nochls] parameter.
Display depth     : 0             /* The stack display depth. Default (0) is to display all stack entries.
Text filename     :               /* The client text trace file
Binary filename   :               /* The client binary trace file
Last status       : 1             /* Last status of mailbox Qio activity (useful if VMS low on
                                  /*  resources and MONMQ fails)
Connection map[0]  : 0            /* A bit map of all connected channels (maximum no. of channels is 128)
```

Closing and deleting an LU

CLOSE [lu=*number*]

CLOSE LU

This command performs the opposite to **OPEN** and closes and deletes the specified LU. The LU is closed in a controlled sequence by first signalling each connected process to disconnect, then resetting each channel and then finally deassigning the trace mailbox and deleting the shared section. This command should only be performed when a trace session has been completed.

Display channel details

SHOW CHANNELS [full] [connected] [chl=*range*]

This command displays the details of the specified channels. The [connected] parameter will cause only channels that have a thread connected to be displayed. For example, the `show channels connected` command displays the following:

```
Chl Pid/Tid      Mailbox   Stack  History  RTime  Time   Mask     Process Name
0   00000245/1   800b0330    4       0       0      0   fffffff    AMQZLAA0.EXE
1   00000244/1   800b0200    8       0       0      0   ffffffff   RUNMQCHI.EXE
2   00000243/1   800b01e0   10       0       0      0   ffffffff   AMQRRMFA.EXE
3   00000242/1   800b01c0    4       0       0      0   ffffffff   AMQZLLP0.EXE
4   00000241/1   800b0220    5       0       0      0   ffffffff   AMQHASMX.EXE
5   00000240/1   800b03f0    4       0       0      0   ffffffff   AMQZXMA0.EXE
```

The [full] parameter displays the complete definition of the specified channels. For example, the **show channels full connected chl=0:3** command displays:

```
Pid/Tid             : 0000024b/1        /* Connected threads process id and thread sequence number
Status              : *** Connected *** /* Current status of channel (thread is connected)
Process name        : AMQRRMFA.EXE      /* Process name of connected thread
Assigned LU         : 0                 /* This channels associated LU
Channel no.         : 2                 /* Allocated channle number within the LU
Mailbox channel     : 800b01e0          /* Connected threads mailbox channel number for the trace mailbox
Current stack depth : 10                /* Threads current stack depth
Circular logging    : Disabled          /* History enabled flag
Next log entry      : 0                 /* Next history buffer slot number
Realtime tracing    : Disabled          /* Real time enable flag (needs client to read messages)
Time stamping       : Disabled          /* Enables timestamping for this threads messages
Trace mask          : ffffffff          /* Hexadecimal format of trace mask for this thread (see show mask command)
Step mode           : Off               /* Not yet implemented
No Wait             : On                /* Forces threads qio activity to wait for a resource if not available
Last QIO status     : 0                 /* Threads last qio call status
Mapped address      : 9a2000-c9ffff     /* The virtual mapped address range of the LU global section for this thread
```

Display the current trace mask for a channel

SHOW MASK [chl=*range*]

This command displays the current trace mask for a channel. A highlighted line indicates that the btrace mask bit is enabled. For example, the command, **show mask chl=1** displays:

```
Trace Mask for Channel 1

Bit 00 - (fent) function entry          Function entry messages
Bit 01 - (fout) function exit           Function exit messages
Bit 02 - (ferr) function exit with error Function exit with error return status
Bit 03 - (fxx) missing function exit    Unbalanced function entry/exit message (see note below)
Bit 04 - (dgn) diagnostic messages      Diagnostic messages
Bit 05 - (shm) shared memory            OVMS shared memory messages
Bit 06 - (spl) spinlocks                OVMS spinlock messages
Bit 07 - (evt) events                   OVMS event messages
Bit 08 - (mtx) mutexes                  OVMS mutex messages
Bit 09 - (prc) process msgs             OVMS thread messages
Bit 10 - (msc) miscellaneous            OVMS kernal niscellaneous messages
Bit 11 - (inf) informational            Internal data messages as requested by show command
Bit 12 -                                Reserved for user defined messages
```

This output shows that function entry, function exit, spinlocks and event messages will be traced for this thread. All other types of messages are blocked.

Display the contents of the target threads stack

SHOW STACK [chl=*range*]

This command displays the contents of the target threads stack. For example, the command **show stack chl=0:1** displays:

```
0001- 00:00:00.00 03 - 01 -->| ExecCtrlrMain
0002 - 12:36:20.18 03 - 02 --->| zcpReceiveOnLink
0003 - 12:36:20.81 03 - 03 ---->| xcsWaitEventSem
0004 - 12:36:20.83 03 - 04 ----->| vms_evt

0001- 00:00:00.00 03 - 01 -->| ExecCtrlrMain
0002 - 12:36:20.18 03 - 02 --->| zcpSendOnLink
0003 - 12:36:20.81 03 - 03 ---->| xcsPostEventSem
0004 - 12:36:20.83 03 - 04 ----->| vms_evt
```

Display active MQSeries related processes and memory usage

SHOW PROCESSES

This command displays all active MQSeries related processes on the current node along with their memory usage. For example, the command **show process** displays:

```
PID      Proc_Name  Image     Process        WS_Size WS_Peak Virt_Peak Gbl_Pg_Cnt Prc_Pg_Cnt Total_Mem

0000023D BKM3_AG    AMQZLAA0  Agent          23152  16576   203776     3616       12960      16576
0000023C BKM3_CI    RUNMQCHI  Run Chan Init  8752   6208    180832     1840       4368       6208
0000023B BKM3_RM    AMQRRMFA  Repository Mgr 11152  8144    185360     2224       5920       8144
0000023A BKM3_CP    AMQZLLP0  Checkpoint     8752   6384    185952     1920       4464       6384
00000239 BKM3_LG    AMQHASMX  Logger         8752   6288    182016     2080       4208       6288
00000238 BKM3_EC    AMQZXMA0  EC             20752  15232   203792     3536       11680      15216
00000128 _FTA4:     MONMQ     MONMQ Utility  8400   8528    198736     2224       3584       5808
```

Displays all messages held in a channel

SHOW HISTORY [chl=*range*]

This command displays all messages held in the channel circular history buffer. Each message is formatted and the output is indented according to the stack depth at which it was generated. For example, the command **show history chl=3** displays:

```
0215 - 12:35:44.52  03 - 02 ---<| zxcProcessChildren
0216 - 12:35:44.55  03 - 02 --->| zxcStartWLMServer
0217 - 12:35:44.57  03 - 02 ---<| zxcStartWLMServer
0218 - 12:35:44.59  03 - 02 --->| zcpReceiveOnLink
0219 - 12:35:44.61  03 - 03 ---->| xcsRequestMutexSem
0220 - 12:35:44.63  03 - 04 ----->| xllSemReq
0221 - 12:35:44.66  03 - 05 ------>| vms_mtx
0222 - 12:35:44.66  03 - 05 .......| vms_mtx  :- Locking BKM3/@ipcc_m_1_10 - timeout: -1
0223 - 12:35:44.70  03 - 06 ------->| vms_get_lock
0224 - 12:35:44.72  03 - 06 -------<| vms_get_lock
0225 - 12:35:44.74  03 - 05 ------<| vms_mtx
0226 - 12:35:44.76  03 - 04 -----<| xllSemReq
0227 - 12:35:44.79  03 - 03 ----<| xcsRequestMutexSem
0228 - 12:35:44.81  03 - 03 ---->| xcsResetEventSem
0229 - 12:35:44.83  03 - 04 ----->| vms_evt
0230 - 12:35:44.83  03 - 04 ......| vms_evt Reset on mailbox BKM3/@ipcc_e_1_2 : tout = -1
0231 - 12:35:44.87  03 - 05 ------>| vms_get_mbx_chan
0232 - 12:35:44.87  03 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_2
0233 - 12:35:44.87  03 - 05 .......| vms_get_mbx_chan Returning key 1a0
0234 - 12:35:44.94  03 - 05 ------<| vms_get_mbx_chan
0235 - 12:35:44.83  03 - 04 ......| vms_evt rc = 0
0236 - 12:35:44.98  03 - 04 -----<| vms_evt
0237 - 12:35:45.00  03 - 03 ----<| xcsResetEventSem
0238 - 12:35:45.03  03 - 03 ---->| xcsReleaseMutexSem
0239 - 12:35:45.05  03 - 04 ----->| xllSemRel
0240 - 12:35:45.07  03 - 05 ------>| vms_mtx
0241 - 12:35:45.07  03 - 05 .......| vms_mtx  :- Unlocking BKM3/@ipcc_m_1_10 - timeout: -1
0242 - 12:35:45.11  03 - 06 ------->| vms_get_lock
0243 - 12:35:45.13  03 - 06 -------<| vms_get_lock
0244 - 12:35:45.16  03 - 05 ------<| vms_mtx
```

This sample output shows the line number within the history buffer, the time the message was generated, the channel number, the stack depth when the message was generated and the name of the function. When the LU was opened, the maximum number of history messages entries was defined. When this buffer is full, MONMQ wraps back to the first entry and overwrites the first and subsequent messages. While tracing, if an FFST is generated, then at the point of failure tracing is disabled for the failing thread. This is to prevent trace messages generated by error routines from filling the buffer. Therefore the last message displayed in the history buffer is the point at which the FFST was generated.

Display all MQSeries related global sections on the current node

SHOW GLOBALS

This command displays all MQSeries related global sections on the current node.

```
MQS1_shm_00000000
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=6128/383
MQS1_shm_01300010
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=1904/119
MQS1_shm_012c000f
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=464/87
MQS1_shm_012c000e
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=4112/514
MQS1_shm_012c000d
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=240/30
MQS1_shm_012c000c
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=528/132
MQS1_shm_012c000b
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=272/51
MQS1_shm_012c000a
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=4112/771
MQS1_shm_012c0009
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=272/51
MQS1_shm_012c0008
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=144/27
MQS1_shm_012c0007
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=528/99
MQS1_shm_012c0006
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=16/2
MQS1_shm_012c0005
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=128/24
MQS1_shm_012c0004
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=1968/492
MQS1_shm_012c0003
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=1904/476
MQS1_shm_012c0002
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=304/76
MQS1_shm_012c0001
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=1904/595
MQS1_shm_01280000
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=16/6
MQS1_shm_fffffffe
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=16/0
MQS1_shm_ffffffff
       (00000000) WRT     DZRO PRM SYS            Pgltcnt/Refcnt=144/63
```

Signals target thread to send mutex table to client trace process

SHOW MUTEX [chl=*range*]

This command signals the target thread to send the contents of its internal mutex table to the client trace process. Note that it is important that the correct trace mask bits are set to enable this type of informational data to be displayed by the client. Bits INF and DGN must be enabled in the trace mask for this channel. (See "Enable or disable mask bit" on page 336.) For example, the command **show mutex chl=2** displays:

```
Mutex Utilisation for Process AMQZXMA0.EXE - Pid 248 ***

0960 - Lock ID: 0100013c - Name: BKM3/@ipcc_m_1_24
0961 - Lock ID: 020006ca - Name: BKM3/@ipcc_m_1_23
0962 - Lock ID: 0b00061e - Name: BKM3/@ipcc_m_1_22
0963 - Lock ID: 0900068e - Name: BKM3/@ipcc_m_1_21
0964 - Lock ID: 0b00032f - Name: BKM3/@ipcc_m_1_20
0965 - Lock ID: 210006eb - Name: BKM3/@ipcc_m_1_19
0966 - Lock ID: 07000742 - Name: BKM3/@ipcc_m_1_18
0967 - Lock ID: 1e000075 - Name: BKM3/@ipcc_m_1_17
0968 - Lock ID: 0c0004dd - Name: BKM3_m_1_45
0969 - Lock ID: 0d00035a - Name: BKM3/@ipcc_m_1_16
0970 - Lock ID: 190000a1 - Name: BKM3/@ipcc_m_1_15
0971 - Lock ID: 1a0005a3 - Name: BKM3/@ipcc_m_1_14
0972 - Lock ID: 14000628 - Name: BKM3/@ipcc_m_1_13
0973 - Lock ID: 130005f3 - Name: BKM3/@ipcc_m_1_12
0974 - Lock ID: 0f0000dc - Name: BKM3_m_1_43
0975 - Lock ID: 02000095 - Name: BKM3_m_1_42
0976 - Lock ID: 2200053e - Name: BKM3_m_1_41
0977 - Lock ID: 020000fc - Name: BKM3_m_1_40
0978 - Lock ID: 31000113 - Name: BKM3_m_1_39
0979 - Lock ID: 02000555 - Name: BKM3_m_1_38
0980 - Lock ID: 2e000389 - Name: BKM3_m_1_37
0981 - Lock ID: 2300011f - Name: BKM3_m_1_36
0982 - Lock ID: 02000109 - Name: BKM3_m_1_35
0983 - Lock ID: 02000327 - Name: BKM3_m_1_34
0984 - Lock ID: 020004a8 - Name: BKM3_m_1_33
0985 - Lock ID: 02000453 - Name: BKM3_m_1_32
0986 - Lock ID: 260007ad - Name: BKM3_m_1_31
0987 - Lock ID: 0200060c - Name: BKM3_m_1_30
```

The data shows the line number in the history file, the mutex name and the system lock id.

Signals target thread to send internal events table to client trace process

SHOW EVENTS [chl=*range*]

This command signals the target thread to send the contents of its internal events table to the client trace process. Note that it is important that the correct trace mask bits are set to enable this type of informational data to be displayed by the client. Bits INF and DGN must be enabled in the trace mask for this channel. (See "Enable or disable mask bit" on page 336.) For example, the command **show events chl=2** displays:

SHOW EVENTS

```
Event Utilisation for Process AMQZXMA0.EXE - Pid 248 ***

1037 - Channel: 000003e0 - Name: BKM3/@ipcc_e_1_19
1038 - Channel: 000003d0 - Name: BKM3/@ipcc_e_1_18
1039 - Channel: 000003c0 - Name: BKM3/@ipcc_e_1_17
1040 - Channel: 000003b0 - Name: BKM3/@ipcc_e_1_14
1041 - Channel: 000003a0 - Name: BKM3/@ipcc_e_1_13
1042 - Channel: 00000390 - Name: BKM3/@ipcc_e_1_12
1043 - Channel: 00000380 - Name: BKM3/@ipcc_e_1_10
1044 - Channel: 00000370 - Name: BKM3/@ipcc_e_1_9
1045 - Channel: 00000360 - Name: BKM3/@ipcc_e_1_8
1046 - Channel: 00000330 - Name: BKM3/@ipcc_e_1_7
1047 - Channel: 00000300 - Name: BKM3/@ipcc_e_1_6
1048 - Channel: 000002f0 - Name: BKM3/@ipcc_e_1_5
1049 - Channel: 000002b0 - Name: BKM3_e_1_11
1050 - Channel: 000002a0 - Name: BKM3_e_1_10
1051 - Channel: 00000290 - Name: BKM3_e_1_9
1052 - Channel: 00000280 - Name: BKM3_e_1_8
1053 - Channel: 00000270 - Name: BKM3_e_1_7
1054 - Channel: 00000260 - Name: BKM3_e_1_6
1055 - Channel: 00000250 - Name: BKM3_e_1_5
1056 - Channel: 00000240 - Name: BKM3_e_1_4
1057 - Channel: 00000230 - Name: BKM3_e_1_3
1058 - Channel: 00000220 - Name: BKM3_e_1_2
1059 - Channel: 00000210 - Name: BKM3_e_1_1
1060 - Channel: 00000200 - Name: BKM3_e_1_0
1061 - Channel: 000001c0 - Name: BKM3/@ipcc_e_1_4
1062 - Channel: 000001b0 - Name: BKM3/@ipcc_e_1_3
1063 - Channel: 000001a0 - Name: BKM3/@ipcc_e_1_2
1064 - Channel: 00000190 - Name: BKM3/@ipcc_e_1_1
1065 - Channel: 00000180 - Name: BKM3/@ipcc_e_1_0
1066 - *** End of data ***
```

The data shows the line number in the history file, the mailbox channel number and the event name.

Signals target thread to send internal mapped shared memory table to the client trace process

SHOW MEMORY [chl=*range*]

This command signals the target thread to send the contents of its internal mapped shared memory table to the client trace process. Please note that it is important that the correct trace mask bits are set to enable this type of informational data to be received by the client. Bits INF and DGN must be enabled in the trace mask for this channel. (See "Enable or disable mask bit" on page 336.) For example, the command **show memory chl=2** displays:

```
     *** Shared Memory Utilisation for Process AMQZXMA0.EXE - pid/tid 248-1 ***

0942 - ShmId: 0248000f - Addr: 011d8000/01211fff - Perm: 950 - Size: 00038530 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/AMQ
0943 - ShmId: 0248000e - Addr: 00fbc000/011bdfff - Perm: 950 - Size: 002005f8 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/AMQ
0944 - ShmId: 0248000d - Addr: 00f1c000/00f39fff - Perm: 950 - Size: 0001d478 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/WLM
0945 - ShmId: 0248000c - Addr: 00cba000/00cfbfff - Perm: 944 - Size: 000405f0 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/HMEMSET.0
0946 - ShmId: 0248000b - Addr: 00c98000/00cb9fff - Perm: 944 - Size: 000205f0 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/Anon005.0
0947 - ShmId: 0248000a - Addr: 00a96000/00c97fff - Perm: 944 - Size: 00200584 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/Anon004.0
0948 - ShmId: 02480009 - Addr: 00a74000/00a95fff - Perm: 944 - Size: 000205f0 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/Anon003.0
0949 - ShmId: 02480008 - Addr: 00a62000/00a73fff - Perm: 944 - Size: 000105f0 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/Anon002.0
0950 - ShmId: 02480007 - Addr: 00a20000/00a61fff - Perm: 944 - Size: 000405f0 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/Anon001.0
0951 - ShmId: 02480006 - Addr: 00908000/00909fff - Perm: 950 - Size: 00001664 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/PLU
0952 - ShmId: 02480005 - Addr: 008f8000/00907fff - Perm: 950 - Size: 0000fff8 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/IPC
0953 - ShmId: 02480004 - Addr: 00802000/008f7fff - Perm: 950 - Size: 000f4838 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/IPC
0954 - ShmId: 02480003 - Addr: 00714000/00801fff - Perm: 950 - Size: 000ec718 Name: /mqs_root/mqm/qmgrs/BKM3/@ipcc/shmem/SUB
0955 - ShmId: 02480002 - Addr: 006ee000/00713fff - Perm: 944 - Size: 000253a8 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/zutSESSAN
0956 - ShmId: 02480001 - Addr: 00600000/006edfff - Perm: 944 - Size: 000ec710 Name: /mqs_root/mqm/qmgrs/BKM3/shmem/SUBPOOL.0
0957 - ShmId: 01280000 - Addr: 005f8000/005f9fff - Perm: 950 - Size: 00000454 Name:                           /var/mqm/errors
0958 - *** End of data ***
```

The data shows the line number in the history file, shared memory id, the virtual mapped address range, the flags used in creating/mapping to the section and the internal MQSeries name given to the section.

Displays active MQSeries components by name and hexadecimal ids

SHOW COMPONENTS

This command displays all active MQSeries components by name and their associated hexadecimal ids. Use these hex ids in other MONMQ show commands such as show functions and select component. For example, the command **show components** displays:

```
00000001 - Data hardening
00000002 - Log management
00000003 - Object Catalogue
00000004 - Queue management
00000005 - Transaction Management
00000006 - Mobile Component
00000007 - Mobile Component
00000008 - Communications
0000000a - Object Authority Manager
0000000b - Logger
0000000d - LQM Kernal
0000000f - Administration App
00000010 - Administration App
00000013 - Command Server
00000014 - Remote queue processor
00000015 - XA Transaction Manager
00000016 - Data Conversion
00000017 - Common Services
00000018 - Common Services (overflow)
00000019 - Application Interface
0000001a - IPCC
0000001b - DCE Support
0000001c - Pluggable Services
0000001d - Agent
0000001e - XA Transaction Manager
0000001f - C++ Layer
00000020 - CLI
00000021 - Z Utilities
00000022 - Execution Controller
00000023 - App. Bindings
00000024 - Service Component
00000025 - Publish/Subscribe
00000026 - MMC Snap-in for Admin
00000027 - Web Administration
00000028 - KYG Services
00000029 - OVMS MQ kernel
```

Display functions within specified component

SHOW FUNCTIONS [comp=*hex*]

This command displays all functions within the specified component. The component must be entered in hex. Use SHOW COMPONENT to display all active MQSeries components. For example, the command **show functions component=0x1f** displays:

ONSTARTUP start

```
00000000 - ImqBinary::copyOut
00000001 - ImqBinary::pasteIn
00000002 - ImqCache::operator =
00000003 - ImqCache::moreBytes
00000004 - ImqCache::read
00000005 - ImqCache::resizeBuffer
00000006 - ImqCache::setDataOffset
00000007 - ImqCache::setMessageLength
00000008 - ImqCache::useEmptyBuffer
00000009 - ImqCache::write
0000000a - ImqDeadLetterHeader::pasteIn
0000000b - ImqDistributionList::openInfoPrepare
0000000c - ImqItem::structureIdIs
0000000d - ImqQueueManager::backout
0000000e - ImqQueueManager::begin
0000000f - ImqQueueManager::commit
00000010 - ImqQueueManager::connect
00000011 - ImqQueueManager::disconnect
00000012 - ImqMessageTracker::setAccountingToken
00000013 - ImqMessageTracker::setCorrelationId
00000014 - ImqMessageTracker::setGroupId
00000015 - ImqMessageTracker::setMessageId
00000016 - ImqObject::close
00000017 - ImqObject::closeTemporarily
00000018 - ImqObject::inquire
00000019 - ImqObject::open
0000001a - ImqObject::openFor
............
```

Activate tracing from the point a process starts

ONSTARTUP [start] [lu=*number*] [chl=*range*]

This command allows tracing to be activated from the point a process starts. When executed, a logical name MQS_DEF_TRACE is defined in the system logical name table and has an equivalent name of the following format: *lu channel*. When any MQSeries process starts, this logical name is checked inside the processes initialization routine and, if present, connects to the specified LU and channel number. If the channel id is already allocated then the next available channel is used. This command is useful when tracing is required during the early phases of MQSeries process/thread creation.

Prevent MQSeries process from tracing immediately from startup

ONSTARTUP [stop]

This command deasssigns the MQS_DEF_TRACE logical from the system logical name table and thus prevents MQSeries processes from tracing immediately from startup.

Connect target thread to specified channel

CONNECT pid *number* [tid=*number*] [chl=*range*]

This command signals the target thread to connect to the specified channel. If no channel is specified then the first available channel is used.

Disconnect target thread to specified channel

DISCONNECT [chl=*range*]

This command signals the target thread to disconnect from the specified channel.

Display real-time trace message written to the LUs trace mailbox

TRACE START [node=*string*]

This command launches a client trace process to display the real-time trace message written to the LUs trace mailbox. The optional node parameter creates a window on the specified node and directs output to that window.

Detach and end current client process

TRACE STOP

This command causes the current client process to detach from the trace mailbox and end. All threads currently writing to this mailbox are disabled from writing messages.

```
MQT> trace stop

Circular buffering has been disabled for process 24d thread 1
Circular buffering has been disabled for process 24c thread 1
Circular buffering has been disabled for process 24b thread 1
Circular buffering has been disabled for process 248 thread 1
Disconnecting thread pid : 24d, tid : 1 from channel 0 ..... OK
Disconnecting thread pid : 24c, tid : 1 from channel 1 ..... OK
Disconnecting thread pid : 24b, tid : 1 from channel 2 ..... OK
Disconnecting thread pid : 248, tid : 1 from channel 3 .....OK

*** Trace ended - no processes connected ***
```

The above output appears on the trace client window.

Specify trace data

SELECT [component] AND/OR [function] OR [*fname*]

This command allows you to specify up to eight combinations of component/functions to be traced. All other trace data are filtered out. Either a function name can specified or a component or a component/function. The selected component/function if valid is written to the filter table. If no entries exist then ALL function component/functions are traced as the default.

Entering the **SELECT** command with no parameters causes the contents of the filter table to be displayed. Against each line of output will be the table index entry and the component and function in hex and the text name of the function. If only a component is entered then all functions within this component are traced. This is shown as 0xffff against the function value.

For example, the command **SELECT** on its own displays:

```
Chl:0 - Cmp/fnc selection criteria
ALL component/functions
```

The following set of commands,

```
MQT>select fname="kill"
MQT>select comp=0x1f
MQT>select comp=0x20 func=0x3
MQT>select
```

displays:

```
Chl:0 - Cmp/fnc selection criteria
Idx: 0 - Cmp: 00000029 - Fnc: 00000005 - Name - kill
Idx: 1 - Cmp: 00000019 - Fnc: 0000ffff - Name -
Idx: 2 - Cmp: 00000016 - Fnc: 00000003 - Name - vqiAddCacheEntry
```

Remove single entry from the trace filter table

DESELECT INDEX=<0:7>

This command removes a single entry from the trace filter table as specified by the table index parameter. All components/functions are traced when all eight entries are empty. For example, a **select** command displays the following entries with their indexes:

```
Chl:0 - Cmp/fnc selection criteria
Idx: 0 - Cmp: 00000029 - Fnc: 00000005 - Name - kill
Idx: 1 - Cmp: 00000019 - Fnc: 0000ffff - Name -
Idx: 2 - Cmp: 00000016 - Fnc: 00000003 - Name - vqiAddCacheEntry
```

The following deselect commands remove the specified processes or functions:

```
MQT> desel index=0
MQT> desel index=2

MQT>select
Chl:0 - Cmp/fnc selection criteria
Idx: 1 - Cmp: 00000019 - Fnc: 0000ffff - Name -
--------------------------------------------
MQT> deselect index=1
MQT> select

Chl:0 - Cmp/fnc selection criteria
ALL component/functions
--------------------------------------------
```

Client process writes trace messages to a binary file

OPEN BINARY [filename=*string*]

This command opens a trace message binary file and causes the client process to write realtime trace messages to this file. This file can be later used for analyzing performance of MQSeries applications. The default filename is `mqs_root:[mqm.errors]mqs_buffer_`*`xx`*`.bin` (where *xx* is the LU number).

Close binary trace messages file

CLOSE BINARY

This command closes the specified LU binary trace file.

Client process writes trace messages to a text file

OPEN TEXT [filename=*string*]

This command opens a readable text file and causes the client process to write formatted binary trace messages to this file. This file can be viewed later by simply using the DCL type command or edit. The default filename is `mqs_root:[mqm.errors]mqs_buffer_`*`xx`*`.lis` (where *xx* is the LU number). The advantages of using this type of output file is that it requires no preprocessing for it to be read. However the disadvantage is that it consumes more disk space than a binary file.

Close text trace messages file

CLOSE TEXT

This command closes the specified LU text trace file.

Timestamp messages

ENABLE TIMESTAMP [chl=*range*]

This command sets the Timestamp flag in the channel definition table. Use this command to force MQSeries processes to stamp each message with the current time. This flag has to be set when using a binary trace file for performance analysis.

Stop timestamping messages

DISABLE TIMESTAMP [chl=*range*]

This command unsets the Timestamp flag in the channel definition table. (See "Timestamp messages".)

Enable tracing

ENABLE TRACE [chl=*range*]

This command sets the RTime trace flag in the channel definition table. Use this flag to enable and disable the sending of trace messages to a trace client. When a thread is connected and a trace client is present, for example, TRACE START can be used to switch the channel in or out rather than disconnecting this thread.

Disable tracing

DISABLE TRACE [chl=*range*]

This command unsets the RTime trace flag in the channel definition table. (See "Enable tracing".)

Save message history

ENABLE HISTORY [chl=*range*]

This command sets the History flag in the channel definition table for the specified channels. This command signals the connected thread to write trace messages to the LUs circular buffer. As the writing of the message is performed by the traced process then it is not necessary for a client process to exist. The size of the trace circular buffer is defined during LU creation by the **open** command. This buffer wraps back to the beginning when the last entry is written. The Next Log record field in the LU definition table specifies where the next record in the buffer is to be written.

Disable message history

DISABLE HISTORY [chl=*range*]

This command unsets the History flag in the channel definition table for the specified channels. See “Save message history”.

Delete message history

DELETE HISTORY [chl=*range*]

This command deletes all messages in the circular history buffer. This command can be performed even when there are processes writing to the buffer so it is not necessary to disable history before deleting.

Set history depth

SET [depth]

This command controls the maximum stack depth to be output to the trace client window. Messages deeper than this value will not be output however they will appear in the binary trace file and history buffer if enabled. The default value of zero allows all messages at whatever stack depth to be output. Users are advised to set this to a very low value (for example, 1) when writing analysis data to the binary file. Full stack display will adversely affect the performance of a client process.

Reset stack and history data for a channel

SET [free] [chl=*range*]

This command resets the specified channel. All existing stack and history data is deleted and the channel is unallocated and available for reuse.

Enable or disable mask bit

SET [mask=*var*] [chl=*range*]

SET mask

This command either enables or disables a mask bit within the connected threads bit mask field. Each bit represents a message type that is generated by an MQSeries process. You can use this command to filter the type of messages that need to be traced. The message types are as follows:

```
MQT>set mask = 0xffffff chl=1
MQT>show mask chl=1
```

```
Trace Mask for Channel 1
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -
```

To specify a combination of these message types delimit each mask type with an OR symbol for example:

```
MQT>set mask =0x0 chl=1 MQT>show mask chl=1
```

```
Trace Mask for Channel 1
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -
```

```
MQT>set mask = mtx | evt | fent chl=1
MQT>show mask chl=1
```

SET mask

```
Trace Mask for Channel 1
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -
```

Each mask comprises of eight mask types which you can toggle to either enable or disable a particular message type. For example if you were interested only in function entry points then enter the command **set mask = fent**.

Set a color for a channel

SET COLOR [chl=*range*]

This command associates a color with the specified channel. All output related to this channel is displayed in this color until either the color is changed or the channel is reset. This command is useful for highlighting or distinguishing between different threads' messages within a single output stream. For example, the commands:

```
MQT> set color=yellow chl=2
MQT> set color=blue chl=0
MQT> sho chan chl=0:3 connected
```

displays:

```
Chl  Pid/Tid     Mailbox  Stack  History  RTime  Time  Mask        Process Name
0   00000245/1   800b0330   4      0       0      0    ffffffff    AMQZLAA0.EXE
1   00000244/1   800b0200   8      0       0      0    ffffffff    RUNMQCHI.EXE
2   00000243/1   800b01e0  10      0       0      0    ffffffff    AMQRRMFA.EXE
3   00000242/1   800b01c0   4      0       0      0    ffffffff    AMQZLLP0.EXE
```

where Channel 0 is blue and Channel 2 is yellow.

Redirect output to file

SET OUTPUT [filename=*string*]

This command directs all output to the specified file and disables output to the display. The **output** command, when used as a parameter with other commands, is effective for that command only. Note that errors continue to be reported to the display device and not to file. Only valid trace data is written to the specified file.

Analyze trace binary file

ANALYSE [component] [function] [unit=*xx*]

ANALYSE

This command analyzes the contents of trace binary file previously used in a trace session. Although you can specify the component or function to be analyzed, this is effective only if the file contains such data relating to this component or function. For example, if when the file was generated, you specified a trace mask or even selected a specific component, then only these selected items are found in the binary file and hence components or functions outside this criteria cannot be used in the analysis.

Note: If you are going to use the full command, spell it ANALYSE with a *S*. Do not confuse this MONMQ command with the OpenVMS command ANALYZE. The two commands are different from each other.

The unit parameter is used to specify the unit of time for the analysis and can have one of the following values (*xx*) - seconds, milliseconds, microseconds, nanoseconds The default is milliseconds.

For example, to display output in microseconds, using the command **analyse unit=micro:** The following is a sample output for this command:

ANALYSE

```
============================================================================================
                          COMPONENT :- Common Services
============================================================================================
Calls  Minimum     Average     Maximum       Total            Function
 42      0.00        66.41      311.78      2789.15        xcsRequestMutexSem
 42      0.00        96.98      222.64      4072.98        xcsReleaseMutexSem
  6      0.00       138.50      286.11       831.00        xcsResetEventSem
  5      0.00     10112.60    10159.51     50563.01        xcsWaitEventSem
  6      0.00       564.74     1029.23      3388.45        xcsCheckExtendMemory
 42      0.00        51.32      266.86      2155.41        xllSemReq
 42      0.00        73.05      159.17      3068.16        xllSemRel
============================================================================================
                          COMPONENT :- Common Services (overflow)
============================================================================================
Calls  Minimum     Average     Maximum       Total            Function
 36      0.00        41.15      122.06      1481.35        xcsCheckProcess
  6      0.00        37.92       68.35       227.52        xihGetConnSPDetailsFromList
  6      0.00        65.26      114.25       391.58        xihHANDLEtoSUBPOOLFn
  6      0.00        12.69       23.44      1267.49        xihGetNextSetConnDetailsFromList
  6      0.00        12.53       22.46        75.19        xcsRequestThreadMutexSem
  6      0.00        12.37       22.46        74.21        xcsReleaseThreadMutexSem
============================================================================================
                          COMPONENT :- IPCC
============================================================================================
Calls  Minimum     Average     Maximum       Total            Function
  5      0.00     10589.97    11029.84     52949.85        xcpReceiveOnLink
============================================================================================
                          COMPONENT :- CLI
============================================================================================
Calls  Minimum     Average     Maximum        Total           Function
  6      0.00         1.95        1.95          11.72       zapInquireStatus
============================================================================================
                          COMPONENT :- Execution Controller
============================================================================================
Calls  Minimum     Average     Maximum        Total           Function
  6      0.00       954.46     3275.18      11726.79        zxcProcessChildren
  6      0.00        12.69       22.46          76.17       zxcStartWLMServer
============================================================================================
                          COMPONENT :- OVMS MQ kernel
============================================================================================
Calls  Minimum     Average     Maximum        Total           Function
 36      0.00        15.49       99.60         557.58       kill
  6      0.00         0.65        3.91           3.91       vms_mapgbl
 84      0.00        11.17       88.16         938.68       vms_get_lock
 84      0.00        42.41      221.94        3562.54       vms_mtx
 12      0.00        44.51      149.40         534.14       vms_get_mbx_chan
 11      0.00      4651.77    10137.05       51169.42       vms_evt
  6      0.00         1.63        4.88           9.76       vms_check_health
```

Columns are:

Calls	The number of entries into the function during the trace session.
Minimum	This is the fastest time spent inside the function.
Average	This is the total time spent inside all calls to the function divided by the number of calls.
Maximum	The longest time spent inside the function.
Total	The total time spent in this function for all calls.
Function	The function name.

Note: The scope of the analysis is the content of the binary trace file. It is up to the user to define the boundaries of the analysis by opening a trace binary file and enable/disabling the trace at the desired time.

Display current state of MQSeries threads

FFST [chl=*range*]

This command forces the thread connected to the target channel to force an FFST. This command does NOT effect the target threads path of execution. This command allows you to take a snapshot of any MQSeries threads current state. The FFST cut contains the threads resource usage, privileges and other useful system information. The FFST is clearly marked in the header as having been created by MONMQ (see below) and is NOT a result of a failure.

The following is some sample output:

```
MQSeries First Failure Symptom Report
  =====================================

  Date/Time         :- Wednesday November 12  10:59:38 GMT 2000
  Host Name         :- CATWMN (Unknown)
  PIDS              :- 5697175
  LVLS              :- 510
  Product Long Name :- MQSeries for OpenVMS Alpha
  Vendor            :- IBM
  Probe Id          :- VM026000
  Application Name  :- MQM
  Component         :- vms_evt
  Build Date        :- Oct 22 2000 (Collector)
  Userid            :- [400,400] (SYSTEM)
  Program Name      :- AMQZXMA0.EXE
  Process           :- 00000248
  Thread            :- 00000001
  QueueManager      :- BKM3
  Major Errorcode   :- xecF_E_UNEXPECTED_SYSTEM_RC
  Minor Errorcode   :- OK
  Probe Type        :- MSGAMQ6119
  Probe Severity    :- 2
  Probe Description :- AMQ6119: An internal MQSeries error has occurred
                         (***   FORCED FFST BY USER ***)
  Comment1          :- *** FORCED FFST BY USER ***
  Comment2          :- -SYSTEM-S-NORMAL, normal successful completion

etc.....
```

Close trace and exit MONMQ

EXIT

This command performs a CLOSE command and exits MONMQ.

Quit MONMQ without closing trace

QUIT

This command does not perform a CLOSE command but exits MONMQ. This command is useful if you want to leave Trace running but want to shutdown MONMQ. The next time MONMQ is activated the previous trace session is resumed.

```
QUIT
```

Managing shared memory with MONMQ

In unusual circumstances, for example, a queue manager failure or forced shutdown with the OpenVMS stop /id command, it is possible that MQSeries shared memory segments will not be automatically deleted by the queue manager. If this occurs it will not be possible to restart the queue manager, because **strmqm** will report that the queue manager is already running.

MONMQ can list MQ shared memory (global sections) that currently exist, and can delete these shared memory sections.

Note: Ensure that all queue managers are shutdown before using the MONMQ utility to delete shared memory segments. Deleting the shared memory of a running queue manager causes the queue manager to fail, possibly corrupting the queue files.

The MONMQ SHOW PROCESS command can be used to ensure that there are no MQ processes running. If there are processes running on a failed queue manager that can not be stopped by the endmqm command then the OpenVMS DCL command stop /id=<pid> can be used.

Check there are no MQSeries processes running by using the following command:

```
MQT> show process
```

```
MQ Processes
PID       Proc Name       Image     Process         WS Size    WS Peak    Virt Peak  Gbl Pg Cnt Prc Pg Cnt Total Mem
--------- --------------- --------- --------------- ---------- ---------- ---------- ---------- ---------- ----------

List the shared memory global sections that currently exist

MQT> show globals
MQS1_shm_2695000a
                  (00000000) WRT    DZRO PRM SYS Pgltcnt/Refcnt=4112/514
MQS1_shm_26950009
                  (00000000) WRT    DZRO PRM SYS Pgltcnt/Refcnt=272/34
MQS1_shm_ffffffff
                  (00000000) WRT    DZRO TMP SYS Pgltcnt/Refcnt=144/63
MQS1_shm_00000000
                  (00000000) WRT    DZRO TMP SYS Pgltcnt/Refcnt=6304/394
```

List the shared memory global sections that currently exist by using the command:

```
MQT> show globals
```

Delete these global sections:

```
MQT> delete
Deleted global section: MQS1_shm_2695000a
Deleted global section: MQS1_shm_26950009
Deleted global section: MQS1_shm_ffffffff
sys$delgbl - unable to delete section MQS1_shm_00000000
```

The error deleting section MQS_shm_00000000 is expected since this section is used by MONMQ. You can now exit MONMQ by issuing the command:

```
MQT> exit
```

You can use the delete command from within a script if you are certain that all queue manager processes are stopped:

```
$ monmq delete
Deleted global section: MQS1_shm_2695000a
Deleted global section: MQS1_shm_26950009
Deleted global section: MQS1_shm_ffffffff
sys$delgbl - unable to delete section MQS1_shm_00000000
```

Scripts and macros in MONMQ

It is possible to run a script of MONMQ commands either from within MONMQ or from the command prompt. Scripts can be useful to collect a set of data, or to configure the MONMQ environment. When MONMQ starts, a script is run from `SYS$MANAGER:MQS_TRACE_STARTUP.MQT` to configure the trace variables in MONMQ.

Note: If the script is not in the current directory, the full path name to the script must be quoted. For example:

```
MQT> ! "sys$manager:test.mqt"
```

It is also possible to define a macro to shorten common or repetitive tasks. A macro declaration consists of three parts:

1. The first part is the macro name, which must be a unique command name.
2. The second part is the macro body, which can span multiple lines and consists of a list of MQSeries commands. The macro body is delimited by { and }.

   The MONMQ prompt changes to **MACRO> when a multiple line macro body is being declared. Any $n, where n is a single digit number, is replaced with parameter n on the macro command line.
3. The third part of a macro definition is a short help text description that is displayed when help <macroname> is used. The help text must be quoted.

You must consider timing issues when declaring a macro. A macro processes very quickly, but some MONMQ commands signal a remote process to perform a task, and this task must be finished before the next macro command is started.

For this reason a short delay is sometimes required. You do this by using the **sleep** command, which has a delay parameter that is specified in tenths of a second.

The following commands can be entered to create a macro that disconnects a channel, resets the trace mask, and frees the channel.

Note: More than one MONMQ command can be placed on a line by using the ";" as a separator.

```
MQT> declare tmpchl intrange "variable to hold a chl range temporarily"
MQT> macro remove { set tmpchl = chl ; dis chl= $1 ; sleep delay=5
**MACRO> set mask=0xffffffff chl= $1 ; set free chl = $1
**MACRO> set chl=tmpchl
**MACRO> } "A macro to disconnect and free channels Param: chl number"
MQT> help remove

VERB remove:
A macro to disconnect and free channels Param: chl number

Macro text:
set tmpchl = chl ; dis chl= $1 ; sleep delay=5 ; set mask=0xffffffff chl=$1
; set free chl = $1 ; set chl=tmpchl
MQT>remove 4
ok - process disconnected process 282 from channel 4
```

Sample trace session

This section describes a typical trace session showing each MONMQ command in sequence. This sample is tracing a running queue manager's execution controller and its related agents main thread.

Before you begin the trace, the following conditions must be met:

- Start queue manager to be traced - STRMQM BKM3
- Check that `sys$manager:mqs_trace_startup.mqt` has no additional commands apart from the preinstalled defaults.
- Check that the logical `MQS_DEF_TRACE` is NOT defined. If it is then perform an **ONSTARTUP END** in MONMQ.

Start momq.

```
>monmq
```

The MONMQ prompt is displayed:

```
MQT>
```

Open a single LU with an ID of zero with ten channels and a history buffer of 100 messages. (Note: 100 messages would be too small for normal tracing purposes. 1000 is normally adequate.)

```
MQT> open lu=0 nochls=10 buffer=100
 ok - LU:0 opened
```

Display LU1 definition:

```
MQT> show seg lu=1
```

```
Trace LU            : 1
Mailbox name         : MQS_TRC_MBX_1
Device name          : MBA431:
Status               : Disabled
Mailbox channel      : 384
History buffer size  : 100
Threads mapped #  : 1
Time stamping        : Enabled
Max channels #       : 10
Display depth        : 0
Text filename        :
Binary filename      :
Last status          : 1
Connection map[0]  : 0
========================================
```

Display MQSeries processes:

```
MQT> show process
```

```
PID        Proc_Name  Image      Process     WS_Size  WS_Peak Virt_Peak  Gbl_Pg_Cnt  Prc_Pg_Cnt Total_Mem

2A00023D   BKM1_AG    AMQZLAA0   Agent         23152  16576   203776     3616         12960     16576
2A00023C   BKM1_CI    RUNMQCHI   Run Chan Init 8752   6208    180832     1840         4368      6208
2A00023B   BKM1_RM    AMQRRMFA   Repository Mgr11152  8144    185360     2224         5920      8144
2A00023A   BKM1_CP    AMQZLLP0   Checkpoint    8752   6384    185952     1920         4464      6384
2A000239   BKM1_LG    AMQHASMX   Logger        8752   6288    182016     2080         4208      6288
2A000238   BKM1_EC    AMQZXMA0   EC            20752  15232   203792     3536         11680     15216
2A000128   _FTA4:     MONMQ      MONMQ Utility 8400   8528    198736     2224         3584      5808
                                                                                     ------------------
                                                                                                52112
```

Identify the execution controller and agent process and connect them to channel one and two respectively.

```
MQT>connect pid=0x238 tid=1 lu=1 chl=1
MQT>connect pid=0x23D tid=1 lu=1 chl=2
```

Check connection details.

```
MQT>show channel full connected lu=1
```

Sample trace session

```
Pid/Tid              : 2a000bc/1
Status               : *** Connected ***
Process name         : AMQZXMA0.EXE
Assigned LU          : 1
Channel no.          : 1
Mailbox channel      : 800c03f0
Current stack depth  : 4
Circular logging     : Disabled
Next log entry       : 0
Realtime tracing     : Disabled
Time stamping        : Disabled
Trace mask           : ffffffff
Step mode            : Off
No Wait              : On
Last QIO status      : 0
Mapped address       : 1242000-126bfff
========================================

Pid/Tid              : 2a000c1/1
Status               : *** Connected ***
Process name         : AMQZLAA0.EXE
Assigned LU          : 1
Channel no.          : 2
Mailbox channel      : 800c03f0
Current stack depth  : 4
Circular logging     : Disabled
Next log entry       : 0
Realtime tracing     : Disabled
Time stamping        : Disabled
Trace mask           : ffffffff
Step mode            : Off
No Wait              : On
Last QIO status      : 0
Mapped address       : 1376000-139ffff
========================================
```

Set defaults for chl, lu and tid to save entering these each time for subsequent commands.

```
MQT>default chl=1:2 lu=1 tid=1
```

Set channel colors so as to distinguish between different trace message. Note that the chl parameter is specified in these two commands because, had the default (1:2) been used then both channels would have been set to yellow and then cyan.

```
MQT>set chl=1 color=yellow
MQT>set chl=2 color=cyan
```

Now show channels.

```
MQT>show channels
```

```
Chl  Pid/Tid     Mailbox  Stack  History  RTime  Time  Mask      Process Name
1   2a0000bc/1   800c03f0   4       0      0      0    ffffffff  AMQZXMA.EXE
2   2a0000c1/1   800c0360   4       0      0      0    ffffffff  AMQZLAA0.EXE
```

Now that both processes have been connected to channels, we can now examine their stacks.

```
 MQT>show stacks
```

```
0001-  00:00:00.00 03 - 01 -->| ExecCtrlrMain
0002 - 00:00:00.00 03 - 02 --->| zcpReceiveOnLink
0003 - 00:00:00.00 03 - 03 ---->| xcsWaitEventSem
0004 - 00:00:00.00 03 - 04 ----->| vms_evt

0001- 00:00:00.00 03 - 01 -->| zlaMain
0002 - 00:00:00.00 03 - 02 --->| zcpReceiveOnLink
0003 - 00:00:00.00 03 - 03 ---->| xcsWaitEventSem
0004 - 00:00:00.00 03 - 04 ----->| vms_evt
```

By enabling timestamping for these two channels we are able to see whether either process has hung or not.

```
MQT>enable timestamp
MQT>show stack
```

```
0001- 00:00:00.00 03 - 01 -->| ExecCtrlrMain
0002 - 12:36:20.18 03 - 02 --->| zcpReceiveOnLink
0003 - 12:36:20.81 03 - 03 ---->| xcsWaitEventSem
0004 - 12:36:20.83 03 - 04 ----->| vms_evt

0001- 00:00:00.00 03 - 01 -->| zlaMain
0002 - 12:36:20.18 03 - 02 --->| zcpReceiveOnLink
0003 - 12:36:20.81 03 - 03 ---->| xcsWaitEventSem
0004 - 12:36:20.83 03 - 04 ----->| vms_evt
```

It can now be seen that there are some messages with a valid timestamp. This shows that both processes are active. In this case both processes are in an event loop with a 10 second timeout period. This timeout can be checked against the message timestamp by continuously performing a **show stack** command until there is a change in the timestamp data.

By enabling history we can now force each process to write their trace messages to the circular buffer.

```
MQT>enable history
MQT> show history
```

At this point we are writing all trace messages to the buffer. You can check this by showing the trace mask and the component/function table.

```
MQT>show mask
```

Sample trace session

```
Trace Mask for Channel 1
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -

Trace Mask for Channel 2
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -
```

```
MQT> select
```

```
Chl:1 - Cmp/fnc selection criteria
ALL component/functions
--------------------------------------------
Chl:2 - Cmp/fnc selection criteria
ALL component/functions
--------------------------------------------
```

```
Trace Mask for Channel 1
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -

Trace Mask for Channel 2
Bit 00 - (fent) function entry
Bit 01 - (fout) function exit
Bit 02 - (ferr) function exit with error
Bit 03 - (fxx) missing function exit
Bit 04 - (dgn) diagnostic messages
Bit 05 - (shm) shared memory
Bit 06 - (spl) spinlocks
Bit 07 - (evt) events
Bit 08 - (mtx) mutexes
Bit 09 - (prc) process msgs
Bit 10 - (msc) miscellaneous
Bit 11 - (inf) informational
Bit 12 -
```

Let's now focus on a particular type of message. Say, for example, that we are interested only in shared memory diagnostic messages.

```
MQT>set mask=shm
MQT>show mask
```

Both processes will now only write diagnostic memory type messages to the buffer. Let's delete the buffer, wait a few seconds and re-examine the contents of the buffer.

```
MQT>clear history

(wait a few seconds)

MQT> show history
```

Sample trace session

```
*** Trace History Chl:1 ***

0990 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0991 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0992 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0993 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0994 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0995 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0996 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0997 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0998 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0999 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff

*** End of buffer ***
*** Trace History Chl: ***


*** End of buffer ***
```

Now let's also display event type diagnostic messages in the trace output. We must wait for a few seconds after setting the mask.

```
MQT>set mask=evt | shm

(wait a few seconds)

MQT>show history
```

```
*** Trace History Chl:1 ***

0977 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0978 - 00:00:00.00 00 - 04 ......| vms_evt Event BKM3/@ipcc_e_1_2 TIMEOUT
0979 - 00:00:00.00 00 - 04 ......| vms_evt rc = 1
0980 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0981 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0982 - 00:00:00.00 00 - 04 ......| vms_evt Reset on mailbox BKM3/@ipcc_e_1_2 : tout = -1
0983 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_2
0984 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Returning key 1a0
0985 - 00:00:00.00 00 - 04 ......| vms_evt rc = 0
0986 - 00:00:00.00 00 - 04 ......| vms_evt Wait on mailbox BKM3/@ipcc_e_1_2 : tout = 10000
0987 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_2
0988 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Returning key 1a0
0989 - 00:00:00.00 00 - 04 ......| vms_evt Event BKM3/@ipcc_e_1_2 TIMEOUT
0990 - 00:00:00.00 00 - 04 ......| vms_evt rc = 1
0991 - 00:00:00.00 00 - 04 ......| vms_mapgbl key : fffffffe - addr : 0/0
0992 - 00:00:00.00 00 - 04 ......| vms_mapgbl Section MQS1_shm_fffffffe mapped at 1510000-1511fff
0993 - 00:00:00.00 00 - 04 ......| vms_evt Reset on mailbox BKM3/@ipcc_e_1_2 : tout = -1
0994 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_2
0995 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Returning key 1a0
0996 - 00:00:00.00 00 - 04 ......| vms_evt rc = 0
0997 - 00:00:00.00 00 - 04 ......| vms_evt Wait on mailbox BKM3/@ipcc_e_1_2 : tout = 10000
0998 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_2
0999 - 00:00:00.00 00 - 05 .......| vms_get_mbx_chan Returning key 1a0

*** End of buffer ***

*** Trace History Chl:2 ***

0982 - 00:00:00.00 01 - 04 ......| vms_evt Event BKM3/@ipcc_e_1_7 TIMEOUT
0983 - 00:00:00.00 01 - 04 ......| vms_evt rc = 1
0984 - 00:00:00.00 01 - 04 ......| vms_evt Reset on mailbox BKM3/@ipcc_e_1_7 : tout = -1
0985 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_7
0986 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Returning key 1c0
0987 - 00:00:00.00 01 - 04 ......| vms_evt rc = 0
0988 - 00:00:00.00 01 - 04 ......| vms_evt Wait on mailbox BKM3/@ipcc_e_1_7 : tout = 10000
0989 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_7
0990 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Returning key 1c0
0991 - 00:00:00.00 01 - 04 ......| vms_evt Event BKM3/@ipcc_e_1_7 TIMEOUT
0992 - 00:00:00.00 01 - 04 ......| vms_evt rc = 1
0993 - 00:00:00.00 01 - 04 ......| vms_evt Reset on mailbox BKM3/@ipcc_e_1_7 : tout = -1
0994 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_7
0995 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Returning key 1c0
0996 - 00:00:00.00 01 - 04 ......| vms_evt rc = 0
0997 - 00:00:00.00 01 - 04 ......| vms_evt Wait on mailbox BKM3/@ipcc_e_1_7 : tout = 10000
0998 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Getting mbx BKM3/@ipcc_e_1_7
0999 - 00:00:00.00 01 - 05 .......| vms_get_mbx_chan Returning key 1c0

*** End of buffer ***
```

```
MQT>disable history
MQT>clear history
```

Now we focus on tracing a specific function. Using **SHOW COMPONENT** and **SHOW FUNCTION** we can identify the particular area we want to trace. In this example we are going to trace the common services function 'xcsRequestMutexSem'. The component is 0x17 and the function code is 0x1b. We can set this one of two ways:

Sample trace session

```
MQT>select comp=0x17 func=0x1b
```

or

```
MQT>select fname="xcsRequestMutexSem"
```

If we now enable history, we find that no output appears in the buffer. This is because we need to reset the trace mask bits to all.

```
MQT>enable history
MQT>show history
```

```
*** Trace History Chl:1 ***

*** End of buffer ***
*** Trace History Ch:2 ***


*** End of buffer ***
```

```
MQT>set mask=0xffffffff
MQT>show history
```

```
*** Trace History Chl:1 ***

0973 - 00:00:00.00 01 - 02 --->| xcsRequestMutexSem
0974 - 00:00:00.00 01 - 03 ---->| xllSemReq
0975 - 00:00:00.00 01 - 04 ----->| vms_mtx
0976 - 00:00:00.00 01 - 04 ......| vms_mtx  :- Locking BKM3_m_1_45 - timeout: -1
0977 - 00:00:00.00 01 - 05 ------>| vms_get_lock
0978 - 00:00:00.00 01 - 05 ------<| vms_get_lock
0979 - 00:00:00.00 01 - 04 -----<| vms_mtx
0980 - 00:00:00.00 01 - 03 ----<| xllSemReq
0981 - 00:00:00.00 01 - 02 ---<| xcsRequestMutexSem
0982 - 00:00:00.00 01 - 03 ---->| xcsRequestMutexSem
0983 - 00:00:00.00 01 - 04 ----->| xllSemReq
0984 - 00:00:00.00 01 - 05 ------>| vms_mtx
0985 - 00:00:00.00 01 - 05 .......| vms_mtx  :- Locking BKM3_m_1_6 - timeout: -1
0986 - 00:00:00.00 01 - 06 ------->| vms_get_lock
0987 - 00:00:00.00 01 - 06 -------<| vms_get_lock
0988 - 00:00:00.00 01 - 05 ------<| vms_mtx
0989 - 00:00:00.00 01 - 04 -----<| xllSemReq
0990 - 00:00:00.00 01 - 03 ----<| xcsRequestMutexSem
0991 - 00:00:00.00 01 - 03 ---->| xcsRequestMutexSem
0992 - 00:00:00.00 01 - 04 ----->| xllSemReq
0993 - 00:00:00.00 01 - 05 ------>| vms_mtx
0994 - 00:00:00.00 01 - 05 .......| vms_mtx  :- Locking BKM3/@ipcc_m_1_18 - timeout: -1
0995 - 00:00:00.00 01 - 06 ------->| vms_get_lock
0996 - 00:00:00.00 01 - 06 -------<| vms_get_lock
0997 - 00:00:00.00 01 - 05 ------<| vms_mtx
0998 - 00:00:00.00 01 - 04 -----<| xllSemReq
0999 - 00:00:00.00 01 - 03 ----<| xcsRequestMutexSem

*** End of buffer ***
```

We can now see that only the specified function and child functions are traced for both processes. Up to eight components and functions can be traced simultaneously using the **select** command. To enable trace in real time (that is, as it happens) we need to create a client process to display the messages for a specific LU. We do this by performing the **TRACE** command.

This launches a client process on the specified node and waits for incoming trace messages. Trace sessions on client windows can still be controlled using MONMQ.

```
MQT>trace start node="mihell"
```

Now enable the client process and display the messages as and when they arrive.

```
MQT>enable trace
```

MQSeries threads can be added or removed from the trace output at will. Threads can remain connected but their trace data can be disabled so that tracing has no adverse effects on performance.

Tracing can be initiated the moment a process or thread starts. The **ONSTARTUP** command is used to do this and results in all new MQSeries processes to be traced from startup.

To shutdown a trace session, all active channels should be disabled and client process ended. The **close** command will do all this for you.

If you want to leave tracing running then use **quit** from MONMQ and resume tracing at a later date.

Note that trace mask bits and component/function selection are very different. Trace mask bits control the output of trace message types. For example trace entry and trace output are message types. If you disable these then whatever you set using the **select** command will have no effect because component/function selection relies on these mask bits being set.

Appendix I. User exits

MQSeries for Compaq OpenVMS supports both channel exit programs and data-conversion exit programs. For information about channel exits, see the *MQSeries Intercommunication* book. For information about data-conversion exits, see the *MQSeries Application Programming Guide* and the *MQSeries Application Programming Reference* book.

This appendix provides information specific to the use of exit programs in MQSeries for Compaq OpenVMS.

Channel and Workload Exits

The requirement to link a separate threaded version of an Exit is not applicable in MQSeries for Compaq OpenVMS.

MQSeries Cluster Workload Exits

When linking a workload exit on OpenVMS, the following should be specified in the linker options file:

```
sys$share:mqm/share
sys$share:mqutl/share
SYMBOL_VECTOR=(clwlFunction=PROCEDURE,MQStart=PROCEDURE)
```

A system wide executive logical name is required to reference the exit image. For example if the exit name is SYS$SHARE:AMQSWLM.EXE the following logical name should be defined:

```
$DEFINE/SYSTEM/EXEC AMQSWLM SYS$SHARE:AMQSWLM
```

The .EXE file extension must not be specified in the logical name definition.

For this logical name to be defined during system startup, define it in SYS$MANAGER:MQS_SYSTARTUP.COM.

Appendix J. Trusted applications

If performance is an important consideration in your environment and your environment is stable, then user applications, channels, and listeners may be defined to be ″trusted″ that is, they use fastpath binding. (The time taken to process MQPUT and MQGET calls of nonpersistent messages can be reduced by up to 400% on OpenVMS systems.)

In a trusted application, the MQSeries application and the local queue manager agent become the same process. The application connects directly to queue manager resources and effectively becomes an extension of the queue manager. This option can compromise the integrity of a queue manager as there is no protection from overwriting its storage.

Also, trusted applications may need to create certain resources like shared memory. These resources may need to be accessed by another queue manager process and, therefore, must be owned by the same UIC. The queue manager processes all run under the MQM account and thus trusted applications must also run under this account.

The issues detailed above should be considered before using trusted applications.

User applications

It is not necessary to run your application directly from the MQM account. Following a successful connection to a queue manager, MQSeries will automatically modify the security profile of the active thread such that the thread assumes the identity of the MQM account. The natural identity of the thread is resumed following a call to disconnect from the queue manager.

It is important to note that while a trusted application is connected to a queue manager the application will be effectively running under the MQM account. If it is necessary to change the identity of the thread to another UIC while connected to a queue manager, you must ensure that you change it back to MQM before making the next MQI call.

Setting up trusted applications

To run a trusted application on MQSeries for OpenVMS you should specify the type of binding in the Options field of the MQCONNX call to be MQCNO_FASTPATH_BINDING. (For standard binding use the MQCNO_STANDARD_BINDING option.) If no options are specified (MQCNO_NONE) the default is to use STANDARD_BINDING.

In addition, the logical name MQ_CONNECT_TYPE may be used to override the binding type specified on the MQCONNX call. If the logical name is defined, it should have the value FASTPATH or STANDARD to select the type of binding required. However, FASTPATH binding is used only if the connect option is appropriately specified on the MQCONNX call. This logical name enables you to execute an application with the STANDARD_BINDING if any problems occur with the FASTPATH_BINDING, without the need to rebuild the application.

In summary, to run a trusted application, either:

- Specify the MQCNO_FASTPATH_BINDING option on the MQCONNX call and define the MQ_CONNECT_TYPE logical name as FASTPATH

or

- Specify the MQCNO_FASTPATH_BINDING option on the MQCONNX call and leave the MQ_CONNECT_TYPE logical name undefined.

For further information on the use of trusted applications see the *MQSeries Intercommunication*.

Running channels and listeners as trusted applications

Channel programs started using the **runmqsc start channel** command run under the MQM account. Channel receiver programs started by incoming TCP (or DECnet connect) requests run under the MQM account also.

The **runmqchl** and **runmqlsr** commands create a detached process that runs under the MQM account. A combination of the MQ_CONNECT_TYPE logical name and MQIBindType in the channels stanza of a queue manager's `qm.ini` file define whether a channel or listener is to be run as trusted.

To set up a trusted channel or listener, either:

- Specify MQIBindType=FASTPATH in the `qm.ini` file and set the logical name to FASTPATH

or

- Specify MQIBindType=FASTPATH in the `qm.ini` file and leave the logical name undefined.

Fast, nonpersistent messages

The nonpersistent message speed (NPMSPEED) channel attribute can be used to specify the speed at which nonpersistent messages are to be sent. You can specify either normal or fast. The default is fast, which means that nonpersistent messages on a channel need not wait for syncpoint before being made available for retrieval. Such messages become available for retrieval far more quickly but may be lost if there is a transmission failure or if the channel stops while the messages are in transit. For further information on running channels and listeners as trusted applications and fast, nonpersistent messages see the *MQSeries Intercommunication* book.

Appendix K. Ancillary information

This appendix lists any ancillary information that you need to setup MQSeries for Compaq OpenVMS.

The information contained in this appendix will be inserted into the identified book, the next time that the book is refreshed.

Application Programming Guide

The information on programming on OpenVMS will be amended to note that Message Queue Interface calls cannot be made from within an AST routine.

The reason for this is that MQSeries uses AST routines itself and these routines cannot run while another AST routine is active.

Application triggering

The command file MQTRIGGER.COM is supplied as an example of a command file designed to take the parameters supplied by the MQSeries trigger monitor (RUNMQTRM) and separate the fields in the MQTMC2 structure.

The command file expects the first parameter to be the image, or command file, to invoke with selected fields from the MQTMC2 structure.

MQTRIGGER passes the following fields from the MQTMC2 structure to the invoked image or command file:

Parameter	MQTMC2 Field
1	QName
2	ProcessName
3	TriggerData
4	ApplType
5	UserData
6	QMgrName

Examples

1. To trigger the amqsech image:

   The ApplicId field of the trigger process definition is specified as follows:

```
APPLICID('@mqs_examples:mqtrigger $mqbin:amqsech')
```

   This example assumes that the MQBIN logical directory has been defined as:

```
SYS$SYSROOT:[SYSHLP.EXAMPLES.MQSERIES.BIN]
```

2. To invoke a command file, dka200:[user]cmd.com:

Ancillary information

The ApplicId field of the trigger process definition is specified as follows:

```
APPLICID('@mqs_examples:mqtrigger @dka200:[user]cmd')
```

Appendix L. 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.

Notices

Trademarks

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, V5.2
- MQSeries for AS/400, V5.2
- MQSeries for AT&T GIS UNIX, V2.2
- MQSeries for Compaq OpenVMS Alpha, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.2
- MQSeries for Linux, V5.2
- 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, V5.2
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Tandem NonStop Kernel, V2.2.0.1
- MQSeries for VSE/ESA, V2.1.1
- MQSeries for Windows, V2.0
- MQSeries for Windows, V2.1
- MQSeries for Windows NT and Windows 2000, V5.2

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 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, V5.2

MQSeries for AIX, V5.0 Quick Beginnings, GC33-1867

MQSeries for AS/400, V5.2

MQSeries for AS/400 V5.1 Quick Beginnings, GC34-5557

MQSeries for AS/400 V5.1 System Administration, SC34-5558

MQSeries for AS/400 V5.1 Application Programming Reference (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 OpenVMS Alpha, V5.1

MQSeries for Compaq OpenVMS Alpha Quick Beginnings, GC34-5885

MQSeries for Compaq OpenVMS Alpha System Administration Guide, SC34-5884

MQSeries for Compaq Tru64 UNIX, V5.1

MQSeries for Compaq Tru64 UNIX, V5.1 Quick Beginnings, GC34-5684

MQSeries for HP-UX, V5.2

MQSeries for HP-UX, V5.0 Quick Beginnings, GC33-1869

MQSeries for Linux, V5.2

MQSeries for Linux Quick Beginnings, GC34-5691

MQSeries for OS/2 Warp, V5.1

MQSeries for OS/2 Warp, V5.0 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 System Administration 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 Version 1.2 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, V5.2

MQSeries for Sun Solaris, V5.0 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, V2.1.1

MQSeries for VSE/ESA™ Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA System Management Guide, GC34-5364

MQSeries for Windows, V2.0

MQSeries for Windows V2.0 User's Guide, GC33-1822

MQSeries for Windows, V2.1

MQSeries for Windows V2.1 User's Guide, GC33-1965

MQSeries for Windows NT and Windows 2000, V5.2

MQSeries for Windows NT, V5.0 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, V5.2
- MQSeries for AS/400, V5.2
- MQSeries for Compaq OpenVMS Alpha, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.2
- MQSeries for Linux, V5.2
- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.2
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Windows NT and Windows 2000, V5.2 (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, V5.2
- MQSeries for AS/400, V5.2
- MQSeries for Compaq OpenVMS Alpha, V5.1
- MQSeries for Compaq Tru64 UNIX, V5.1
- MQSeries for HP-UX, V5.2
- MQSeries for Linux, V5.2

- MQSeries for OS/2 Warp, V5.1
- MQSeries for OS/390, V5.2
- MQSeries for Sun Solaris, V5.2
- MQSeries for Sun Solaris, Intel Platform Edition, V5.1
- MQSeries for Windows NT and Windows 2000, V5.2
- 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 an MQSeries SupportPac™.

Related publications

- *Compaq OpenVMS Performance Management*, January 1999

  This book provides information to help you optimize performance on OpenVMS systems.
- *Compaq OpenVMS System Management Utilities* 2 volumes, January 1999

  These books contain reference information for system management utilities with OpenVMS.
- *Character Data Representation Library, Character Data Representation Architecture, Reference and Registry*, SC09–2190–00

  This document provides an overview of Character Data Representation Architecture (CDRA), and defines the elements of the architecture in the form of a reference manual.
- *DecNet SNA Gateway for Synchronous Transport Installation (OpenVMS)*, November 1993

  This guide explains how to install and configure DecNet SNA Gateway.
- *Digital SNA APPC/LU6.2 Programming Interface for OpenVMS*, May 1996

  This guide explains how to install and configure SNA APPC/LU6.2.
- *Digital TCP/IP Services for OpenVMS Installation and Configuration*, January 1999

  This guide provides instructions for installing and configuring Digital TCP/IP.
- *Guidelines for OpenVMS Cluster Configurations*, January 1999

  This guide describes how to maximize OpenVMS cluster availability and scalability.
- *Introduction to Compaq Networking and Data Communications*, (Compaq Part No. 093148)

  This guide provides an overview of Compaq networking and data communications concepts, tasks, products, and manuals.

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* book.

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 and Windows 2000, 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*.

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 and Windows 2000, 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 OS/2 Warp, and MQSeries for Windows NT and Windows 2000, 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 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 and Windows 2000, 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.

D

DAE. Dump analysis and elimination.

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 Windows 2000, 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. See also *browse*.

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

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

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*.

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.

.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 and Windows 2000, 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 and Windows 2000, 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.

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.

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.

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 and Windows 2000, 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 and Windows 2000, 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 and Windows 2000, 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. See also *persistent message* and *nonpersistent message*. 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. Contrast with *programmable command format (PCF)*.

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 and Windows 2000, 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 and Windows 2000, 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 and Windows 2000, 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.

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 and Windows 2000, a term used for a user identifier. Used by the object authority manager for checking authorizations to system resources.

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.

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.

Resource Recovery Services (RRS). An OS/390 facility that provides 2-phase syncpoint support across participating 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*.

RRS. Resource Recovery Services.

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.

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 and Windows 2000, 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

TACL. Tandem Advanced Command Language.

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). See also *global trace* and *performance trace*.

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.

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 44–1962–842327
  - From within the U.K., use 01962–842327
- 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.

SC34-5884-00