IBM MQSeries for UnixWare

SC33-1379-03

User's Guide

Version 1 Release 4.1

Note!
Before using this information and the product it supports, be sure to read the general information under “Notices” on page xi.

Fourth Edition (January 1997)

This edition applies to Version 1 Release 4.1 of IBM MQSeries for UnixWare (part number 63H9503, program number 5697-265) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

In Europe, Middle East and Africa, use the program number to order the product. Otherwise, order the product by part number.

This book is based on Version 1 Release 4, order number SC33-1379-02. Changes from that edition are marked by vertical lines to the left of the text.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

At the back of this publication is a page titled “Sending your comments to IBM”. If you want to make comments, but the methods described are not available to you, please address them to:

IBM United Kingdom Laboratories, Information Development
Mail Point 095, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom

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

Contents

Figures

Tables

Notices

The following paragraph does not apply to any 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 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.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

Any additional information necessary to achieve interoperability of the programs described in this book with other programs is available from:

The MQSeries Manager
Mail Point 161
IBM United Kingdom Laboratories
Hursley Park
Winchester
Hants
SO21 2JN
U.K.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing to The IBM Director of Licensing, IBM Corporation, 500 Columbus Ave, Thornwood, New York, 10594, U.S.A.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	IBM	MQ
MQSeries	MVS/ESA	OS/2
OS/400	VSE	

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About this book

The purpose of this *User's Guide* is to provide all information necessary for a user to install IBM MQSeries for UnixWare software, as well as how to fully use its features to provide the communications framework for distributed applications based on the IBM's Message Queue Interface (MQI).

To accomplish this goal, this guide describes the IBM MQSeries for UnixWare software—its installation, configuration, and operations—and the programming interface to be used by the developers of applications.

Throughout this document, IBM MQSeries for UnixWare is referred to simply as MQSeries System.

Who should use this book

The introductory product description sections of this guide will be of interest to *all users*. Beyond that, different portions of this guide are intended for these different audiences:

- *System* or *Network Administrators* responsible for installing, operating and maintaining the MQSeries System software will be primarily interested in Chapter 2, "Installation" on page 5 through Chapter 5, "System operation" on page 59.
- *Distributed Application Designers* will be interested in Chapter 3, "Planning" on page 21 through Chapter 6, "Application programming interface" on page 107.
- *Application Developers* will be primarily interested in Chapter 6, "Application programming interface" on page 107.

What's in this book

The guide provides information about the MQSeries System software as implemented for UNIX systems.

How to use this book

This *User's Guide* consists of six chapters and four appendices organized as follows:

- *Chapter 1, "Product description" on page 1* - Describes the MQSeries System and services, provides an overview of the components and architecture, and provides an application example.
- *Chapter 2, "Installation" on page 5* - Highlights the system requirements for using the MQSeries System software and provides a detailed procedure for installing the software.
- *Chapter 3, "Planning" on page 21* - Provides an overview of the considerations for implementing a distributed application using the MQSeries System.
- *Chapter 4, "Configuration" on page 29* - Covers the details for creating the system services to support your application.
- *Chapter 5, "System operation" on page 59* - Provides procedures for activating system services and troubleshooting system problems.
- *Chapter 6, "Application programming interface" on page 107* - Provides a reference of the Application Programming Interface (API) calls.
- *Appendix A, "UnixWare error messages" on page 157* - Lists the full set of error messages built into the MQSeries System software.
- *Appendix B, "Sample source listings" on page 167* - Illustrates the use of each of the MQI calls.
- *Appendix C, "C programming language examples" on page 183* - Provides examples of how to invoke message-queuing calls.
- *Appendix D, "Configuration worksheets" on page 189* - Contains blank worksheets to aid in the design and planning of a distributed application using the MQSeries System.

Typographical conventions

boldface
Identifies an item in an MQSeries System window. The item could be a keyword, an action, a field label, or a pushbutton. Whenever one of the steps in a procedure includes a word in boldface, look for an item in the window that is labeled with that word.

bold italics
Are used for emphasis. ***Take extra care*** wherever you see bold italics!

italics
Identify one of the following:

- New terms that describe MQSeries System components or concepts. A term printed in italics is usually followed by its definition.
- Parameters for which you supply the actual names or values.
- References to other books.

<angle brackets>
Identify a key on the keyboard. The instruction "press <Enter>" means "Find the key labeled 'Enter' and press it." If the instruction identifies two (or more) keys, hold down the first key while you press the second key.

`monospace`
Identifies one of the following:

- Text you must type as shown, ensuring you type the uppercase and lowercase characters exactly.
- Names of files and directories (path names).

Where to find more information

MQSeries publications

Evaluating products

IBM MQSeries Brochure, G511-1908

IBM MQSeries: An Introduction to Messaging and Queuing, GC33-0805

IBM MQSeries: Concepts and Architecture, GC33-1141

IBM MQSeries Message Queue Interface Technical Reference, SC33-0850

Planning

IBM MQSeries Planning Guide, GC33-1349

IBM MQSeries for MVS/ESA Version 1 Release 1.4 Licensed Program Specifications, GC33-1350

IBM MQSeries for OS/400 Version 3 Release 1 (and later) Licensed Program Specifications, GC33-1360 (softcopy only)

Administration

IBM MQSeries Programmable System Management, SC33-1482

IBM MQSeries Command Reference, SC33-1369

IBM MQSeries for AIX Version 2 Release 2.1 System Management Guide, SC33-1373

IBM MQSeries for AT&T GIS UNIX Version 2.2 System Management Guide, SC33-1642

IBM MQSeries for HP-UX Version 2 Release 2.1 System Management Guide, GC33-1633

IBM MQSeries for MVS/ESA Version 1 Release 1.4 Program Directory, GC33-1626

IBM MQSeries for MVS/ESA Version 1 Release 1.4 System Management Guide, SC33-0806

IBM MQSeries for OS/2 Version 2.0.1 System Management Guide, SC33-1371

IBM MQSeries for OS/400 Version 3 Release 1 (and later) Administration Guide, SC33-1361

IBM MQSeries for OS/400 Version 3 Release 6 Programmable Command Formats, SC33-1228

IBM MQSeries for SunOS Version 2.2 System Management Guide, GC33-1772

IBM MQSeries for Sun Solaris Version 2.2 System Management Guide, GC33-1800

IBM MQSeries for SINIX and DC/OSx Version 2.2 System Management Guide, GC33-1768

IBM MQSeries for Windows NT Version 2 Release 0 System Management Guide, SC33-1643

Application programming

IBM MQSeries Application Programming Guide, SC33-0807

IBM MQSeries Application Programming Reference, SC33-1673

IBM MQSeries Application Programming Summary, SX33-6095

IBM MQSeries for OS/400 Version 3 Release 1 (and later) Application Programming Reference (RPG), SC33-1362

IBM MQSeries for OS/400 Version 3 Release 1 (and Release 6) Application Programming Reference (C and COBOL), SC33-1363

Problem determination

IBM MQSeries for MVS/ESA Version 1 Release 1.4 Problem Determination Guide, SC33-0808

IBM MQSeries for MVS/ESA Version 1 Release 1.4 Messages and Codes, SC33-0819

IBM MQSeries Version 1 Products for UNIX Operating Systems Messages and Codes, SC33-1754

Special topics

IBM MQSeries Distributed Queuing Guide, SC33-1139

IBM MQSeries Clients, GC33-1632

Other MQSeries publications

For information about other MQSeries platforms, see the following publications:

IBM MQSeries for AT&T GIS UNIX User's Guide, SC33-1437

IBM MQSeries for Digital VMS VAX User's Guide, SC33-1144

IBM MQSeries for HP-UX User's Guide, SC33-1376

IBM MQSeries for OS/400 User's Guide, SC33-1145

IBM MQSeries for SCO UNIX User's Guide, SC33-1378

IBM MQSeries for SunOS User's Guide, SC33-1377

IBM MQSeries for Sun Solaris User's Guide, SC33-1439

IBM MQSeries for Tandem NonStop Kernel, SC33-1755

IBM MQSeries for UnixWare User's Guide, SC33-1379

IBM MQSeries for VSE/ESA User's Guide, SC33-1142

Chapter 1. Product description

MQSeries for UnixWare enables application programs to exchange messages with other MQSeries applications running on UNIX or other systems such as IBM Mainframes, VAXs**, Tandems**, etc.

The MQSeries System provides a set of messaging and queuing services which support data transfer between distributed applications. These services allow applications to communicate without knowledge of the lower levels of the communications network and without specific knowledge of the location of the other applications. The messaging and queuing services are accessed via an *application programming interface* (API) which conforms to the IBM Message Queuing Interface (MQI) specification.

Version 1 MQSeries System elements

There are four key conceptual elements within the MQSeries System which must be well understood. They are *messages*, *queues*, *queue manager*, and *channels*.

Messages

All data transferred by the MQSeries System is in the form of a *message* exchanged between cooperating distributed applications. Every message has two parts. The *body* of the message contains the *user data* supplied by an application. This user data is never touched by the MQSeries System.

Ancillary data commonly called a *header,* is added to the message by the MQSeries System to provide routing and other control information required for message delivery. The header is not normally seen by the application programs.

Messages are exchanged between applications via *queues*.

Queues

A message queue is simply a disk file used by the MQSeries System to hold messages. The physical management of queues is entirely hidden from the application programs. Applications have no access to the queues other than through the Message Queuing Interface (MQI).

Message queues are classified as either *local* or *remote.* These terms are defined from an application perspective. A *local queue* is any queue residing on the same message queuing system as the application. A *remote queue* is any queue residing on another message queuing system.

The special case of a local queue which is used to hold messages to be transmitted to another system is called a *transmission queue.*

An *alias queue* is not a true physical queue, but rather a logical naming capability which allows an alias queue name to be resolved to another real queue, either local or remote. This provides a mechanism for logical indirection which often proves a convenient method to allow application programs to be completely independent of the underlying message queuing definitions.

The physical management of the queues is provided by the *queue manager*.

Queue manager

The queue manager is responsible for providing the message queueing services used by applications. Applications access these services by using the MQI calls to communicate with the local queue manager (the queue manager on the same system as the application). It is most common to think of a queue manager as having a one-to-one correspondence to an MQSeries System installation. That is, normally there is one queue manager per system.

Channels

A *channel* is a unidirectional point-to-point communications link between two MQSeries systems. Messages flow over a channel in one direction only. If two MQSeries systems need to *exchange* messages, then two channels are required.

For outbound channels, the MQSeries System reads messages from the associated *transmission* queue and sends them to the remote system via the communications channel. For inbound channels, the MQSeries System receives messages from the communication link and writes them to the destination *local* queue.

Software components of the MQSeries System

The MQSeries System system consists of the following software components:

Message queuing interface (MQI)

The Version 1 MQSeries System implementation of MQI is built around the standard C language function call interface. It is responsible for handling user application requests to read and write from the queuing system, and for arbitrating among multiple requests to the same queue. The MQI functions are provided in the form of an object library. Appropriate MQI functions are integrated into application programs that wish to use MQSeries services.

Message channel agent (MCA)

The Message Channel Agent (MCA) is an executable program which moves messages between machines. It implements the Message Channel Protocol (MCP). The MCP is the high level protocol used to transport messages between MQSeries systems. This protocol is implemented on top of an industry standard Transport Layer Protocol (TLP). The underlying TLP is not provided with the MQSeries System but is a prerequisite.

Message channel agent maintenance daemon (MCAMD)

The Message Channel Agent Maintenance Daemon (MCAMD) is a daemon process required by the MCA (Message Channel Agent). It runs on the same communication server as the MCA(s). The MCAMD provides a centralized Channel Database service allowing MCA(s) and the MQM interface to access and modify the Channel Database. The MCAMD must be started prior to running an instance of MCA or performing any channel configuration tasks.

Message queue management (MQM)

The Version 1 MQSeries System Administration and Operations functions are provided through a menu-driven, screen-oriented program called MQM**.** This program allows the system administrator to define, modify, and delete MQSeries queues, aliases, and channels; and to perform various maintenance tasks such as resetting message sequence numbers, purging queues, and monitoring the status of the MQSeries System.

Sample programs

Source code for three sample application programs is provided. These are simple test programs which will be used in verifying the system installation and which may also be referred to for examples of MQI calls.

The MQSeries System's distributed architecture on UNIX

On a UNIX platform, particularly one which resides on a TCP/IP network, there are three possible styles of operation.

First, all four programs already defined (the MCA, MCAMD, MQM and the user's Application Program, incorporating the MQI) can reside on the same machine. This machine can then communicate with all other machines using MQSeries message queuing. These can be other UNIX machines, mainframes, etc.

Second, it is possible to create a network architecture where one UNIX machine acts as gateway to a mainframe or other systems. In this example, there may be many UNIX machines running several applications which use message queuing. These each would send messages to the gateway machine (a similar UNIX platform). The gateway machine, in turn, would send those messages to the mainframe or other systems. In this example, each machine has all four programs mentioned in the previous paragraph.

Third, it is possible to configure several machines so that the queue storage and administration are common to all of them. In this configuration, all but one of the machines contain only user application programs incorporating the MQI. These machines can be referred to as "MQI application only" systems, because they depend on a remote file service. For more information, refer to "Multiple message queue managers on the same LAN" on page 31.

One of the machines in this configuration contains the MCA and MCAMD programs as well as the MQM administration programs. (It may also contain other application programs incorporating the MQI.) This machine can be referred to as the "comm server."

As mentioned above, in this third configuration, the queue storage is shared between all the "MQI application only" systems and the comm server. This is accomplished by means of a distributed file system (such as NFS) which provides file storage which is addressable from each of the machines.

Note: We do not recommend using NFS across bridges or routers.

All of the "MQI application only" systems and the comm server in a particular configuration must be running the same UNIX platform (for example, all running UnixWare**). A mixture of UnixWare and other UNIX platforms (such as HP-UX**) in the same "MQI application only" configuration is not supported. If you have a mixture of such platforms, you must configure at least one comm server for each platform, so that each "MQI application only" system can be associated with a comm server running the same platform. Messages can be exchanged between the comm servers using channels in the usual way.

It is strongly recommended that all NFS file storage used by your applications, including that used by the MQSeries System, be backed up in the usual way.

Some implementations of NFS do not support record-level locking. If the NFS system which you are using does not support record-level locking, you must not have more than one application getting messages from any one queue at the same time. To ensure that this does not happen, applications in such systems should never use the `MQOO_INPUT_SHARED` option when opening a queue. You should also be aware that in these systems, all `MQGET`, `MQPUT`, and `MQPUT1` calls, even when directed at different queues, are effectively serialized, and this may have a significant effect on performance and throughput. With NFS systems which do support record-level locking, only `MQGET`, `MQPUT`, and `MQPUT1` calls to the same queue are serialized. NFS, as supplied with UnixWare Application Server SDK V1.1 and 2.1, does support record level locking.

Note: An `MQGET` call which is waiting for a message does not hold any locks, and so does not prevent other calls from proceeding while it is waiting. Note also that any call which is unable to obtain a necessary lock will block until it can do so; in particular, an `MQGET` call does not give up if it is still blocked when the wait interval (if one was specified) expires.

Chapter 2. Installation

This chapter provides the installation procedure for the MQSeries System and details software and hardware requirements. The chapter also lists the files and directories created during installation, and describes the means of verifying a successful installation of the MQSeries System.

In the UNIX environment, the MQSeries System has a *distributed architecture* (described in Chapter 1, “Product description” on page 1). The prerequisites for a UNIX machine running the MQSeries System depend on the machine’s role in the Network File Services (NFS) environment.

Normal operation, which is not dependent upon a distributed file system, does not assume the use of NFS.

Prerequisites for normal operation

Combined file/communications server

Hardware and software

- Any 386 DX PC or better
  - Minimum system memory = 16 MB
  - Minimum system disk space = 2 MB + Size of Queues

    Note: For an explanation of system disk space requirements see “System disk space requirements for the MQSeries System” on page 45.
- Any LAN adapter

  with
- UnixWare Application Server SDK 1.1 or later 1.x

  including:
  - TCP/IP
    Customers who wish to use UnixWare across bridges using TCP/IP and who are using UnixWare Application Server V1.1 or V1.1.1 or who are experiencing difficulties should contact their SCO** service representative for further information about the availability of patches: ODICOR.TAR and TOKEN.TAR.

  or
- UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1

  including:
  - TCP/IP

For SNA connectivity

- Any 486 PC or better (including an ISA bus)
  - Minimum system memory = 16 MB
  - Minimum system disk space = 26.5 MB + 2 MB + Size of Queues

    Note: For an explanation of system disk space requirements see “System disk space requirements for the MQSeries System” on page 45.
- Apertus Technologies Inc ELC Adapter (ISA)

  with

  For UnixWare Application Server SDK 1.1 or later 1.x:
  - Express 2.04b. This product is shipped on 1/4 inch tape.

  or
  - Express 2.1.1. This product is shipped on 1/4 inch tape.

For UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1:
  – Express 2.1.2. This product is shipped on 1/4 inch tape.

or

- Emulex Adapter (ISA)

  with

  For UnixWare Application Server SDK 1.1 or later 1.x:
  – Express 2.04b. This product is shipped on 1/4 inch tape.

  or
  – Express 2.1.1. This product is shipped on 1/4 inch tape.

  For UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1:
  – Express 2.1.2. This product is shipped on 1/4 inch tape.

  or

- Madge Token Ring Adapter (ISA)

  For UnixWare Application Server SDK 1.1 or later 1.x:

Madge Card	Express Version
Madge Smart 16/4 AT card part # 52-07	Express 2.04b or Express 2.1.1
Madge Smart 16/4 AT PLUS card part # 52-03	Express 2.1.2

  For UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1:
  – Express 2.1.2 and the Madge Smart 16/4 AT PLUS card part # 52-03.

  Express is shipped on 1/4 inch tape.

File server only

Hardware

- Any 386 DX PC or better
  – Minimum system memory = 16 MB
  – Minimum system disk space = Size of Queues

  Note: For an explanation of system disk space requirements see “System disk space requirements for the MQSeries System” on page 45.

- Any LAN adapter

Software

- UnixWare Application Server SDK 1.1 or later 1.x

  including:

  – TCP/IP
  Customers who wish to use UnixWare across bridges using TCP/IP and who are using UnixWare Application Server V1.1 or V1.1.1 or who are experiencing difficulties should contact their SCO service representative for further information about the availability of patches: ODICOR.TAR and TOKEN.TAR.

  or

- UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1

  including:

  – TCP/IP

- Appropriate LAN software, for example, NFS to match TCP/IP
  – For customers who plan to use NFS, please contact your service representative to obtain all available patches.

Communications server only

Hardware and software

- Any 386 DX PC or better
  - Minimum system memory = 16 MB
  - Minimum system disk space = 2 MB

    Note: For an explanation of system disk space requirements see “System disk space requirements for the MQSeries System” on page 45.
- Any LAN adapter

  with
- UnixWare Application Server SDK 1.1 or later 1.x

  including:
  - TCP/IP
    Customers who wish to use UnixWare across bridges using TCP/IP and who are using UnixWare Application Server V1.1 or V1.1.1 or who are experiencing difficulties should contact their SCO service representative for further information about the availability of patches: ODICOR.TAR and TOKEN.TAR.

  or
- UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1

  including:
  - TCP/IP

SNA connectivity

- Any 486 PC or better (including an ISA bus)
  - Minimum system memory = 16 MB
  - Minimum system disk space = 26.5 MB + 2 MB

    Note: For an explanation of system disk space requirements see “System disk space requirements for the MQSeries System” on page 45.
- Apertus Technologies Inc ELC Adapter (ISA)

  with

  For UnixWare Application Server SDK 1.1 or later 1.x:
  - Express 2.04b. This product is shipped on 1/4 inch tape.

  or
  - Express 2.1.1. This product is shipped on 1/4 inch tape.

  For UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1:
  - Express 2.1.2. This product is shipped on 1/4 inch tape.

  or
- Emulex Adapter (ISA)

  with

  For UnixWare Application Server SDK 1.1 or later 1.x:
  - Express 2.04b. This product is shipped on 1/4 inch tape.

  or
  - Express 2.1.1. This product is shipped on 1/4 inch tape.

  For UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1:
  - Express 2.1.2. This product is shipped on 1/4 inch tape.

  or
- Madge Token Ring Adapter (ISA)

For UnixWare Application Server SDK 1.1 or later 1.x:

Madge Card	Express Version
Madge Smart 16/4 AT card part # 52-07	Express 2.04b or Express 2.1.1
Madge Smart 16/4 AT PLUS card part # 52-03	Express 2.1.2

For UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1:
- Express 2.1.2 and the Madge Smart 16/4 AT PLUS card part # 52-03.

Express is shipped on 1/4 inch tape.

MQI application only

Hardware

- Any 386 DX PC or better
  - Minimum system memory = 16 MB
  - Minimum system disk space = Normal disk space supplied with machine

    Note: For an explanation of system disk space requirements see “System disk space requirements for the MQSeries System” on page 45.
- Any LAN adapter

Software

- UnixWare Application Server SDK 1.1 or later 1.x

  including:
  - TCP/IP
    Customers who wish to use UnixWare across bridges using TCP/IP and who are using UnixWare Application Server V1.1 or V1.1.1 or who are experiencing difficulties should contact their SCO service representative for further information about the availability of patches: ODICOR.TAR and TOKEN.TAR.

  or
- UnixWare Personal Edition 2.01 or later 2.x or Application Server 2.01 or 2.1

  including:
  - TCP/IP
- Appropriate LAN software, for example, NFS to match TCP/IP
  - For customers who plan to use NFS, please contact your service representative to obtain all available patches.

Languages for application development

- C

Prerequisites for NFS operation

In addition to the software and hardware identified above, if you wish to use NFS to store messages on a file server (which also contains the message movement programs), then you will need the following:

Software

NFS software (same version level on both the Queuing Application machine and the Message Movement machine).

Hardware

An appropriate LAN adapter on both the Queuing Application machine and the Message Movement machine.

Contents of distribution media

The distribution media for IBM MQSeries for UnixWare contains the following directories:

- Executable images of all the MQSeries System programs and sample programs.
- C header files for use in applications by developers.
- The MQI object library — shared and static versions.
- Source code for makefile and the sample programs provided with the MQSeries System.

MQSeries System installation

To install the MQSeries System, use the standard UNIX system administration installation utility `pkgadd`.

Pre-Installation

Symbolic links to MQSeries System executables are added to `/usr/bin`. Verify that `/usr/bin` exists in your `PATH` environment variable so that symbolic links to MQSeries System executables can be added. Symbolic links to the Transact message file are also added to the `/var/mqi` directory.

1. Either log in as `root` or use `su`.
2. Terminate or kill all MCAs and the MCAMD.
3. Insert the MQSeries System distribution media, and type:

   ```
   pkgadd -d diskette1 mqi
   ```

   – `mqi` is the name of the package to install.

   – `pkgadd` checks to make sure this product has not already been installed. If the package is already installed, you will be prompted to first execute the package remove command (`pkgrm mqi`). Be sure to note the Queue Manager Config Path name before you remove the package; you will most likely want to restore this value after you install the new package and run `MQM` the first time. Alternatively, you can first save a copy of the `SYSTEM.mgr` file and copy it back to the new `/var/mqi` after installing the new package.

   – When configuring the Queue Manager, the `/var/mqi` directory is used for the queue manager configuration file and log files. If this directory is not present, it is automatically created. If it is ever deleted, it must manually be created again.
4. Enter the Transmission Control Protocol (TCP) port number for the MQSeries System server (that is, `mqmcamd`). This number must be unique and greater than `1024`. Default is `7711`.

   – This port is used by the Message Channel Agent (`MCA`) Maintenance Daemon (`MCAMD`) for communication with `MQM` and `MCA`s.

   – The service is added to the `/etc/services` file as the `mqmcamd` port.
5. The directory name where the Queue Manager file will be stored during its configuration is always `/var/mqi`.
6. Enter the directory name to install the MQSeries System executables and files. Default is `/usr`. Documentation refers to this directory as *install_dir*. All files will be under `install_dir/mqi`.
7. Verify the installation information on your screen and, if correct, press **<ENTER>.** Otherwise, press **<r>** to rerun the `pkgadd` utility or **<q>** to quit installation. A copy of this information will be written to `/var/configure`.

Notes: It is important that the default value of the environment variable LANG be correct. As shipped, the MQSeries System assumes this value to be En_US (United States English). Modify the default value of this variable in /etc/environment.

Once the MQSeries System files have been installed, the user should take the time to read the Service History file (HISTORY in `install_dir/mqi/install`). This file contains valuable information about the software level, service history, special instructions and notes on new features.

Post-Installation

After successfully installing the MQSeries System on your UnixWare platform, carry out the following:

- Run `MQM` to create the queue manager configuration (see "Verifying the MQSeries System installation" on page 10).
- If using SNA/LU6.2, configure the EXPRESS SNA Server to support the transport connections required (see the *EXPRESS SNA Server Configuration Guide*).

When installing a new level of the product (a new version, release, or any maintenance), if the MCAMD is normally invoked by the `init` process, make sure that you first stop the MCAMD process.

If you wish to use NFS to support file service, you must also do the following (refer to "Multiple message queue managers on the same LAN" on page 31):

▼ *Configuring MQSeries System where the MCA and MCAMD reside*

1. Export the following directories via NFS:
   - `/var/mqi`
   - The user-defined directory that was specified during installation (*install_dir*).
   - The directory defined in the queue manager's *config.path* field.

▼ *Configuring MQSeries System where the queuing application resides*

1. Import the following directories via NFS:
   - `/var/mqi`
   - The user-defined directory that was specified during installation (*install_dir*).
   - The directory defined in the queue manager's *config.path* field.
2. Make sure the `mqmcamd` TCP/IP service name and port number as specified in step 4 of the installation is added to this system and the MCA Hostname field on the Queue Manager screen is defined.

Note: The `MCAMD` process needs to be running at all times for MQSeries System programs to operate. Please see "The MCAMD process" on page 102 for a discussion of this topic.

Verifying the MQSeries System installation

The installation verification test illustrates how to create a local queue and use the example programs `zmqread` and `zmqwrite` (see Appendix B, "Sample source listings" on page 167) to send and receive messages using this queue. Create the queue using the Message Queue Manager (MQM) administration screens, as follows:

▼ *Creating an MQSeries System queue*

1. Go to the *install_dir/mqi*`bin` directory.

   Your `PATH` environment variable should include `.` or the full pathname of this directory. Note also that the NLSPATH environment variable must contain `/var/mqi/%N` in order to have access to error messages.
2. At the system prompt, type:

   ```
   mqm
   ```

3. The first time you use MQM, the Queue Manager Definition screen is displayed:

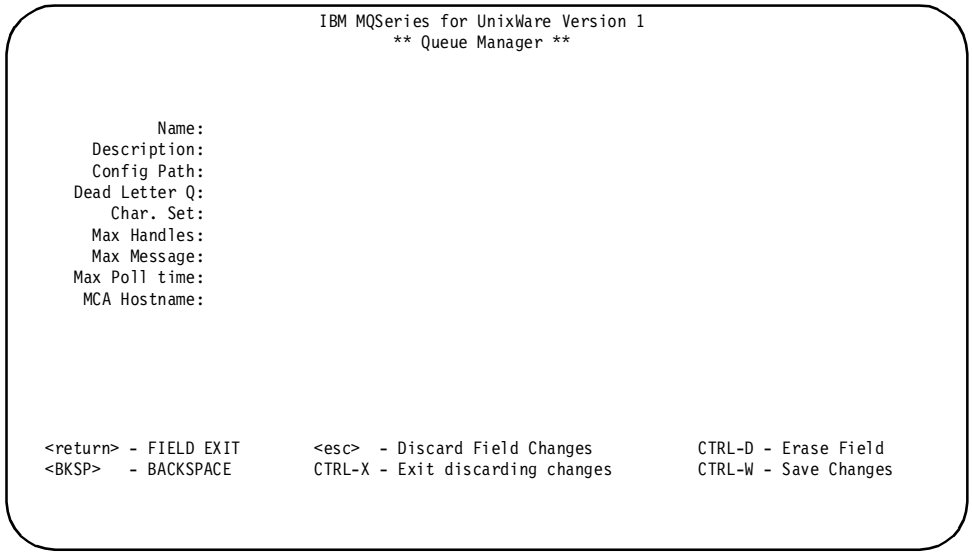


Figure 1. Queue manager definition screen

4. The fields on this screen must be completed before continuing with the installation test. (See “Modify queue manager” on page 61, for a description of the fields.)
5. To display the Main Menu, press **<Ctrl-W>**:

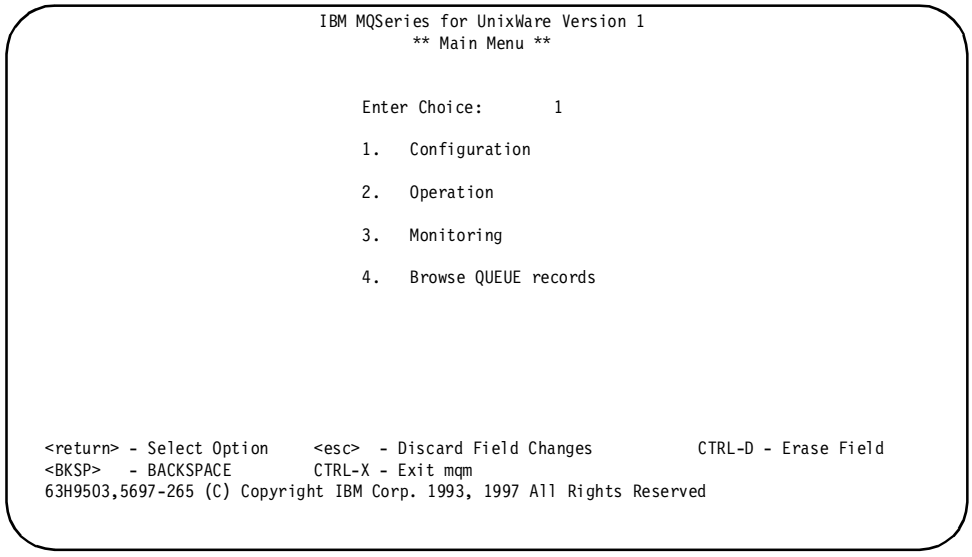


Figure 2. Main menu

6. To select Configuration, type 1 and **<ENTER>** to display the Configuration menu:

```
                          IBM MQSeries for UnixWare Version 1
                               ** Configuration Menu **

                               Enter Choice:       1

                               1.   Modify Queue Manager
                               2.   Display Queue Manager

                               3.   Create Queue
                               4.   Modify Queue
                               5.   Delete Queue
                               6.   Display Queue

                               7.   Create Channel
                               8.   Modify Channel
                               9.   Delete Channel
                               10.  Display Channel


<return> - Select Option    <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Go to previous menu
```

Figure 3. Configuration menu

7. To create a queue, type 3 and **<ENTER>** to display the Define Queue Name menu:

```
                          IBM MQSeries for UnixWare Version 1
                               ** Define Queue Name **

        Queue Type: L       L=Local, R=Remote, AQ=Alias Queue
                            AM=Alias Queue Manager
                            AR=Alias Reply Queue

        Name: ANYQ









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 4. Define queue name

8. For a local queue, fill in the following fields with these values:

   Queue Type: L

   Queue Name: ANYQ

9. To save the entry, press **<Ctrl-W>**. The Create Local Queue screen appears:

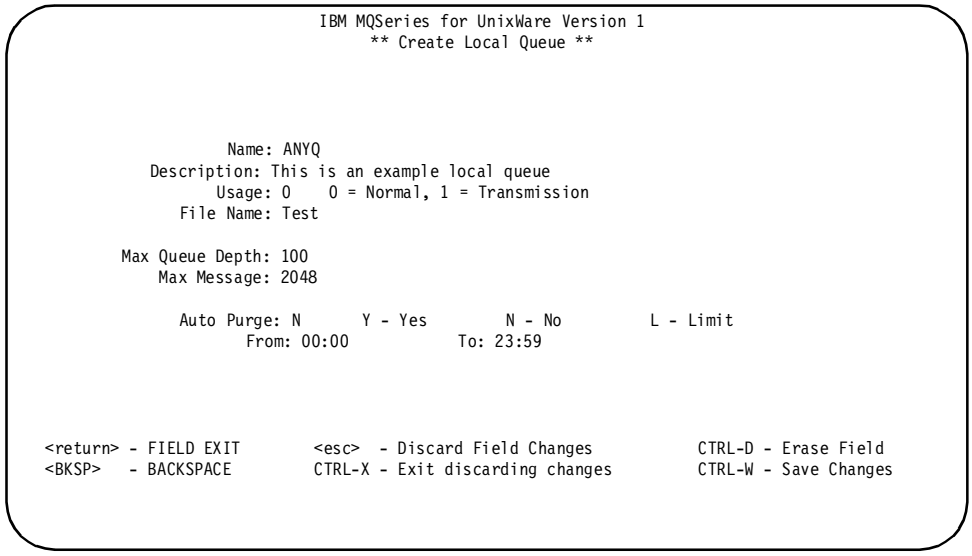


Figure 5. Create local queue

10. Fill in the fields with these values:

Description:	`Anything you like.`
Usage:	`0 (Normal)`
File Name:	`Test`
Max Queue Depth:	100
Max Message:	2048

11. To save the entry, press **<Ctrl-W>**.
12. Press **<Ctrl-X>** to return to the Configuration menu. To display your Queue Definition, type `6` and **<ENTER>**. A selection screen appears. Use the `J` and `K` keys to select the local queue and press **<ENTER>**.
13. The screen that appears displays the queue parameters just entered. Visually verify the correct data has been entered.

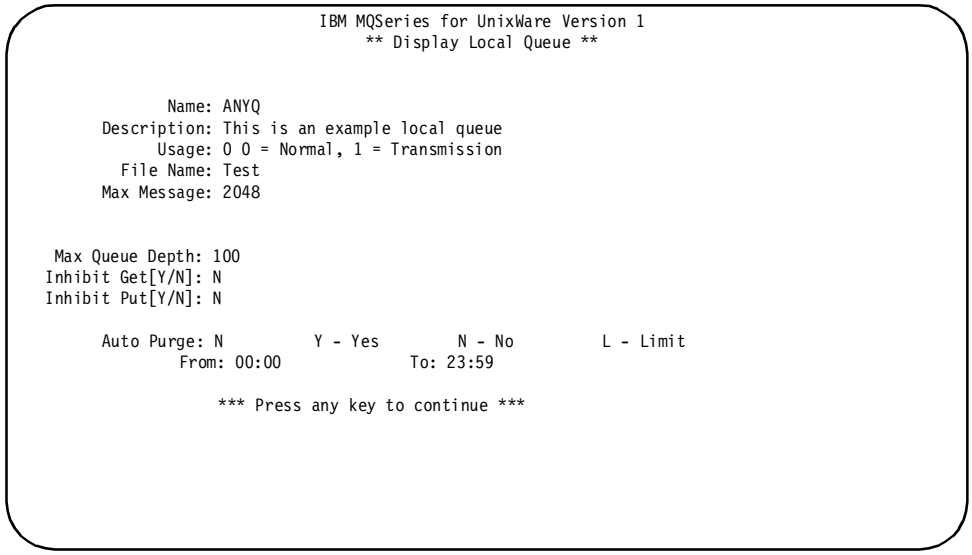


Figure 6. Display local queue

Congratulations. You have created your first MQSeries System queue.

14. Repeatedly press **<Ctrl-X>** to return to the Main Menu without exiting MQM.

▼ *Using zmqwrite and zmqread*

With the local queue that was just created in the first procedure, the following procedure uses the sample program `zmqwrite` to send messages to `zmqread`.

1. From the Main Menu (as shown in Figure 2 on page 11), type `3` and **<ENTER>** to select the Monitoring option and to display the Monitor Menu:

```
                          IBM MQSeries for UnixWare Version 1
                                  ** Monitor Menu **


                              Enter Choice:       1

                              1.   Monitor Queue

                              2.   Monitor Channel




<return> - Select Option     <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Go to previous menu
```

Figure 7. Monitor menu

2. From the Monitor Menu, type `1` and **<ENTER>** to select Monitor Queue.
3. The Queue Monitor screen appears, showing `ANYQ` as the only defined queue. Note the number of messages currently on the queue (DEPTH).

```
                          IBM MQSeries for UnixWare Version 1
                              ** Monitor Local Queues **

Queue                               Type      USERS     LWRIT     DEPTH     G P
================================================================================
ANYQ                                LOCAL     00000     00000     00000     A A









J       - Down                      K         -   Up                  <return>  - Select
CTRL-F  - PgDn                      CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 8. Monitor local queues

4. Move to another UNIX window, and at the system prompt, type:

   ```
   zmqwrite UNIXQMGR#ANYQ 10 1000 "HELLO"
   ```

   `zmqwrite` sends the specified messages addressed to `ANYQ` for the queue manager `UNIXQMGR`. The number of messages sent is displayed on the screen periodically.

5. Return to the window running the MQM Menus.

   The Queue Monitor screen will still be showing `ANYQ` as the only defined queue. Note the number of messages on the queue now.

```
                              IBM MQSeries for UnixWare Version 1
                                  ** Monitor Local Queues **

Queue                                 Type      USERS     LWRIT     DEPTH     G P
================================================================================
ANYQ                                  LOCAL     00000     00010     00010     A A









J       - Down                  K         -   Up                  <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 9. Messages on monitor local queue

6. Move to another UNIX window, and at the system prompt, type:

   ```
   zmqread UNIXQMGR#ANYQ 10
   ```

   `zmqread` reads the specified messages from `ANYQ`. The messages are displayed on the screen as they are read.

```
==>[0001]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0002]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0003]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0004]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0005]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0006]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0007]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0008]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0009]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
==>[0010]:HELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHELLOHEL
```

Figure 10. Reading messages from local queue with zmqread

7. Return to the window running the MQM Menus.

The Queue Monitor screen still displays ANYQ as the only defined queue. Note the number of messages on the queue now.

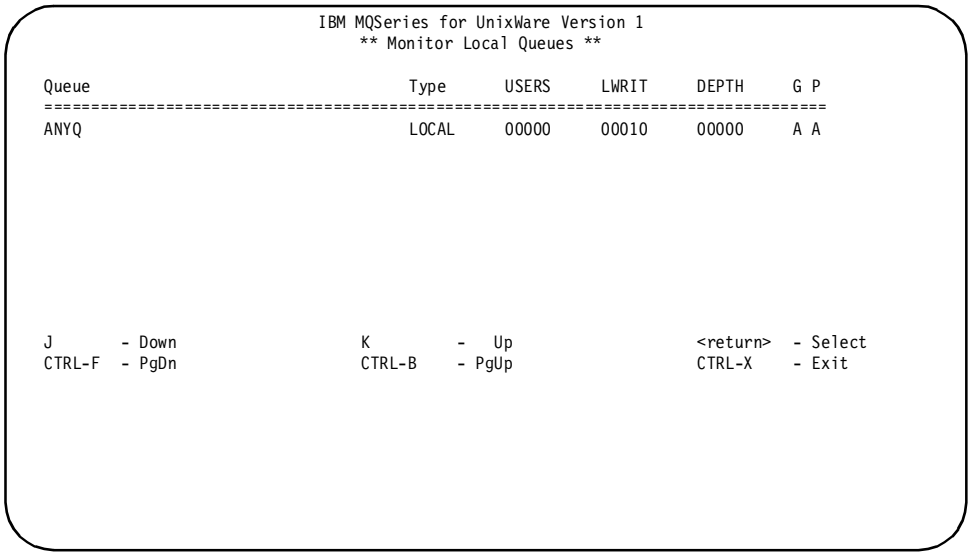


Figure 11. No messages on monitor local queue

The screen displays the number on the queue to have decreased to zero, and the total number of messages written to the queue (LWRIT) remains the same.

8. Exit MQM by pressing **<Ctrl-X>**.

   You have now completed a *local* installation verification test demonstrating that two applications can send/receive messages via an MQSeries System queue. Realize that this test has not tested communications links that connect your system to a remote system.

Notes: In order to expand this test to include a remote link, three steps are required.

1. Install the prerequisite hardware and software required to support the selected transport protocol. Refer to the manufacturers directions for this installation.
2. Define the desired the MQSeries System channel(s). Refer to Chapter 5, “System operation” on page 59 and coordinate with the remote system administrator to accomplish this.
3. Configure the transmission queue(s) and remote queue(s) required for MQSeries System to communicate over the channel.

MQSeries System files and directories

After the MQSeries System has been installed on your *UnixWare* platform, verify the existence of these files and directories..

Table 1. Important files and directories

Path Name/File	Purpose
/usr/bin	Directory for binary files. This directory should be included in user's PATH environment variable specification.
mqm	Symbolic link to install_dir/mqi/bin/mqm
mca	Symbolic link to install_dir/mqi/bin/mca
mcamd	Symbolic link to install_dir/mqi/bin/mcamd
/usr/lib	Directory for library files.
libmqi.so	Symbolic link to shared MQI API library install_dir/mqi/lib/libmqi.so
libmqi.a	Symbolic link to static MQI API library install_dir/mqi/lib/libmqi.a
/var/mqi	This is the base directory for the MQSeries System Queue Manager. It contains the Queue Manager Database and the sub-directory log. It is created automatically by the installation script, though it must be manually created should it ever be deleted.
SYSTEM.mgr	Queue Manager Database. This file needs to be seen by all MQSeries System configuration processes, as well as user programs linked with the MQSeries System MQI library. It is created and maintained by the MQM program.
transact.cat	Default for the MQSeries System message catalog. Used only when the user's native language message catalog cannot be located. This file exists as a symbolic link to install_dir/mqi/install/transact.cat.
/var/mqi/log	Directory where log files are stored. Though created by the installation script, this directory may need to be manually created should it ever be deleted. None of the MQSeries System programs or libraries will take care of this directory creation.
mqi??.log	Files fitting this naming format are error log files. The '?' meta-characters are replaced with the date on which errors were generated. If a log file that is in use is deleted, then all error reporting by active applications will not be viewable.
install_dir/mqi	Base directory to which all the MQSeries System programs are installed through pkgadd.

Table 1. Important files and directories (continued)

Path Name/File	Purpose
`install_dir/mqi/bin`	Directory where the MQSeries System management, transport, and demonstration executables are installed.
`mqm`	Message Queue Management program. Most of the MQSeries System configuration is performed through this utility.
`mca`	Message Channel Agent.
`mcamd`	Message Channel Agent Maintenance Daemon.

`zmqecho`
`zmqread`
`zmqwrite`
	Demonstration programs.

`install_dir/mqi/include`	Directory containing the MQSeries System C header files.

`cmqc.h`
`mqconst.h`
`mqtypes.h`
	The end user need only include `cmqc.h` into the application. The other two will automatically be included by `cmqc.h`.

`install_dir/mqi/src`	Directory containing sample MQSeries System applications.

`zmqecho.c`
`zmqread.c`
`zmqwrite.c`
	These contain the source code that created `zmqecho`, `zmqread` and `zmqwrite` in the `bin` directory.

`makefile`	`makefile` places the executables in `install_dir/mqi/src`.
<*config.path*>	This variable represents the configuration path defined in the Queue Manager configuration screen. This is the path to all queue configuration files as well as queues. This directory must exist prior to specification in the Queue Manager. It must not be deleted once specification has occurred. If it is, then it must be manually recreated or the product will not function.
`SYSTEM.cdb`	Channel Data Base. This file is created by `MCAMD`. Though configuration occurs through `MQM`, the `MCAMD` must be running before `MQM` will allow configuration of channels. Be sure to save a backup copy after making any changes to this file.
`SYSTEM.idx`	Index for all configured queues. This file is created by `MQM`. `MQI` functions need this index file to know where the next available message is for all local queues. Values maintained for specific queues are reevaluated each time the queue is opened without already being in use. Should it be deleted, all queue information will be unobtainable.
`SYSTEM.qdt`	Queue definition table. This file is created by `MQM`. Should it be deleted, all queue information will be lost.

Table 1. Important files and directories (continued)

Path Name/File	Purpose
*.que	Queue files. These are the physical queues. The queues are made up of static-sized records, each record being of the size specified in the local queue creation screen. Since the record size is static, care should be taken when assigning queue sizes. As an example, if messages written to a queue are only 10 bytes, and the queue record size is 1024, 10 records will occupy ((1024 + header_size) * 10) bytes, meaning that 10140 bytes of disk space will be wasted.
*.alt	Temporary queues created by the Purge Deleted Message option on the Operation Menu in MQM.
*.qul	Queue user lock files. These files are used for queue user accounting. If a queue's associated .qul file is deleted, it will be impossible to tell if the queue is currently in use. If not in existence, this file will be recreated by the MQOPEN call.
install_dir/mqi/lib	Directory where the MQSeries System libraries are stored.
libmqi.so	MQSeries System shared library containing all queuing facilities. This is the recommended library to link with.
libmqi.a	MQSeries System library containing all queuing facilities. This is a static library that will be linked into the user's application. Should a patch be made to the library, the user's application will need to be recompiled.
install_dir/mqi/install	Directory where installation scripts and miscellaneous files are stored.
transact.cat	MQSeries System message catalog. A symbolic link to this file exists in the /var/mqi directory.
HISTORY	MQSeries System Service History file. This file contains important information about the MQSeries System, including the version and revision numbers, changes made to the system, and special instructions. It should be read after installing the MQSeries System.

Configuring MQSeries System

The MQSeries System software has now been loaded and the installation has been locally verified using the provided test programs. You can now use the administrative programs and the MQI libraries. But, before user applications may effectively use the system for message transmission, the MQSeries System must be fully configured. Configuration is explained in the following three chapters:

- Chapter 3, “Planning” on page 21, summarizes the planning for new installations.
- Chapter 4, “Configuration” on page 29, provides the configuration guidelines.
- Chapter 5, “System operation” on page 59, describes the MQSeries System administration screens used in the configuration.

Permissions

A user id and group id should be defined for MQSeries operations so that only users in that group may run `MCAMD`, `MCA` or `MQM`. The use of permissions will help prevent unauthorized users from entering `MQM` and performing actions against the MQSeries System processes. The ownership and mode of the executables should be set with the following commands:

- `chown <user_ID> <filename>`
- `chgrp <group_ID> <filename>`
- `chmod u+rwx,g+rwx,o-rwx <filename>`

Service history file

The Service History file, named `HISTORY`, is located in the installation directory and contains the software level, the service history of the product, and any special instructions.

Chapter 3. Planning

This chapter provides an overview of the considerations for implementing a distributed application using the MQSeries System. This chapter will present an overall framework for the planning of a distributed application and will expand on areas specific to the MQSeries System.

A planning framework for distributed applications

As the term “middleware” suggests, the MQSeries System supports the creation of message-enabled applications, and resides between distributed applications and the underlying communications network. As such, it is imbedded in an often long process of planning and implementation.

Several disciplines are involved in this planning. These may be administered independently, resulting in separate but related planning domains for applications, systems, networks, etc., or, they may be integrated to a higher level of planning for the distributed environment. In either case, planning and implementation procedures will vary substantially from one organization to another. Yet, it is often desirable to have a frame of reference when discussing individual planning activities. It is for this purpose that a generic Distributed Planning Procedure is outlined below.

Tasks and responsibilities

Figure 12 identifies tasks and allocates them to the individual or organization typically responsible. In the paragraphs that follow, each of the individual tasks is summarized. Those that include the MQSeries System are expanded further.

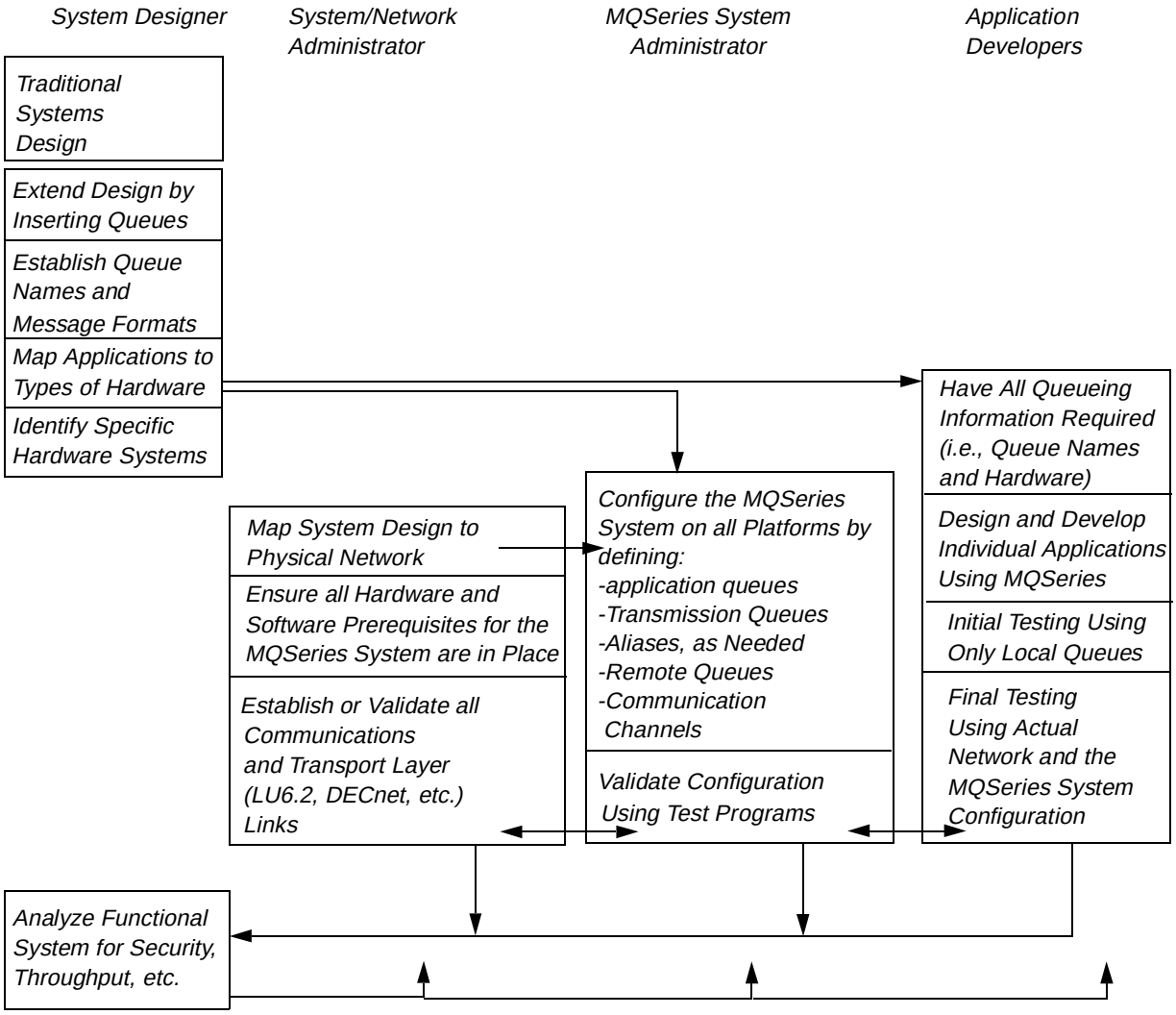


Figure 12. Tasks and responsibilities

System designer tasks

Traditional analysis and design

For new development, the design begins normally. Several well recognized methodologies exist for approaching the basic system design effort. Any one of these results in a functional decomposition of the overall system into a mesh network of processes depicting the flow of information through the designed system. The mesh may be arbitrarily complex based on the system requirements, but each process will be defined in terms of its local function and in terms of *data formats* exchanged with other processes.

For example:

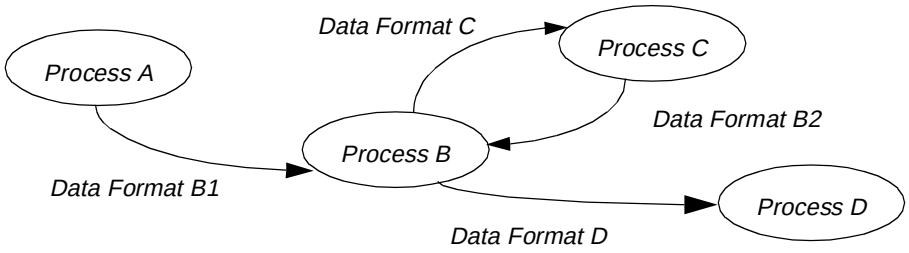


Figure 13. Typical data flow diagram

For existing systems which are to be modified to operate in a distributed environment, the above process may already be complete, or may have never been performed. If documentation at this level is not available, it must be created. The functional decomposition must be accomplished at least to the level that will identify each process which is a candidate for relocation. Each of the processes must be understood in terms of the *data formats* exchanged with other processes.

Extending to a distributed design

In order to extend a “traditional” design to a distributed environment, using messaging and queuing, there are a few essential steps.

- ***Identify which application processes are to be distributed.*** This might apply to all component processes or a subset of the entire system. In many cases, this is a simple step since the primary system goal will have been stated in terms of a desire to distribute a particular function.
- ***Isolate each such process by inserting queues*** (in the design) between it and the remainder of the system.
- ***Assign names to each of the required queues.*** These names are the logical names which will be used by applications throughout the distributed environment to address the queues. It is convenient to think of the queue name as a *logical destination address* for a message. So, the names should be associated with the process which will *receive* messages via that queue.
- ***Define message formats for the new queues*** to replace the exchanged *data formats* in the original design. For example, notice the queues isolating **Process C** in the diagram below:

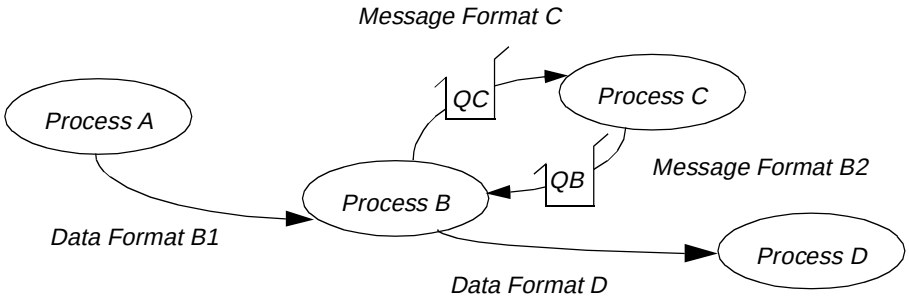


Figure 14. Process C queue isolation

Mapping the design to the physical world

From a purist perspective, the highest level distributed design is complete. However, before much work can be done beyond the design, it must be mapped onto the physical distributed environment. This occurs in two steps.

Map the component processes (applications) to the "type" of host hardware on which they will be implemented. Such as Mainframe, VAX, PC, etc.

At this point, all information is known that is required by the application developers to begin development of individual applications. They know:

- The platform on which the application will run, and, therefore, the MQSeries implementation specifics for that platform.
- The queue(s) on which the application will receive messages.
- The queue(s) to which the application will send messages (i.e., the queues on which destination applications will receive messages).
- The formats of messages to be exchanged via the above queues.

The developers have no need for further knowledge of the underlying network or of the MQSeries System configuration. They specifically have no need to know where destination queues will eventually reside.

Map the component processes (applications) to the "specific" host hardware on which they will be operational. Such as: the Mainframe in Chicago, the VAX called "Mickey" in Engineering, the PC LAN serving the third floor, the UNIX system at IP address 255.25.2.5, etc.

The naming conventions for these systems will be different for every company. In any case, whether formal or informal, these host systems will already be known to the enterprise network or they will have to be added to the network.

This step constitutes the last step of the distributed design and is the highest level map of the logical design to the physical network.

System / Network administrator tasks

Map the logical design to the physical network

This is the detailed extension of the last design step. Verify or complete the map to specific hardware systems. (Completion of the map may be particularly necessary in the case of LAN implementations. The System Designer may not have low level LAN configuration knowledge, such as which workstations are served by which file servers.)

Ensure that hardware and software are in place

Verify that the hardware and software prerequisites for the MQSeries System are installed at each system involved in the distributed implementation.

Establish the transport layer of the network

This is a critical step which requires detailed system/networking knowledge of each platform but very little knowledge of the MQSeries System. This includes:

- Verify that physical communications links (paths through the network) exist between each of these systems.
- Establish any required transport layer definitions (LUs, PUs, NCP Gens, etc.) which are needed to support a logical point-to-point connection between the MQSeries systems.

Some interaction with the MQSeries System administrator is required to complete the above steps. The information shared between the Systems/Network Administrator(s) and the MQSeries System Administrator(s) include:

- End points of point-to-point logical links.
- Number of links between systems.
- Transport protocol used.
- Transport specific names (LU Names, XIDs, Node Names, etc.)

MQSeries System administrator tasks

The MQSeries System Administrator is the focal point for a successful implementation since this is the one activity which touches all others. Interaction will be required with the System Designers, the Network Administrators, and the Application Developers.

The Administrator will have certain operational responsibilities after the distributed system has been implemented, but ***by far the most significant duty is the initial configuration of the MQSeries System queues***. This is the critical function which ties together the underlying transport network and the distributed applications. Briefly, this includes:

- Configuring the MQSeries System Message Queue Manager
- Configuring MQSeries System Local Queues
- Configuring MQSeries System Transmission Queues
- Configuring MQSeries System Queue Aliases
- Configuring MQSeries System Remote Queue Definitions
- Configuring MQSeries System Communications Channels

From a planning perspective, it should be realized that the queue configuration will require some level of coordination throughout the network, but will be accomplished on each individual system. Further, it should be recognized that configurations are typically built in three phases. These phases correspond to:

- ***Test configuration(s)*** to allow local testing of applications using queues, or initial testing of communications lines.
- ***Functional configuration*** to include all communications channels and all queues. This configuration allows full application functional testing. It has not been optimized for performance or been modified for any security or other installation specific requirements.
- ***Operational configuration*** which is an extension of the above after considering performance requirements, security requirements, etc.

Details of all configuration activities are provided in Chapter 4, “Configuration” on page 29, of this document.

Application developer tasks

Application development on individual platforms can begin relatively early in the implementation process. It can start as soon as the developers know:

- The platform on which the application will run, and, therefore, the MQSeries implementation specifics for that platform.
- The queue(s) on which the application will receive messages.
- The queue(s) to which the application will send messages (i.e., the queues on which destination applications will receive messages).
- The formats of messages to be exchanged via the above queues.

The developers have no need for further knowledge of the underlying network or of the MQSeries System configuration. They specifically have no need to know where destination queues will eventually reside.

Development will proceed very much like traditional applications development. The only difference is the use of the API to interface to queues.

The API and other Application design considerations are described in detail in Chapter 6, “Application programming interface” on page 107, of this document.

Including legacy applications in distributed designs

Legacy applications are commonly old, not well understood, not well documented, but they work. So, no one wants to touch them.

Such applications present an obvious paradox when they form a critical piece of a system that is to become distributed. *Their input/output interfaces cannot be readily altered yet they must be modified to support messages and queues*.

The solution is simple. The Legacy application is “sandwiched” between a *preprocess* and a *post-process* application which convert queues and message formats to/from existing data formats used by the legacy software.

To illustrate this, consider a segment of our earlier flow diagram:

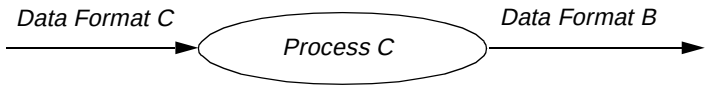


Figure 15. No messaging and queuing

If Process C can be directly modified to use the MQI to take advantage of messaging and queueing, then the result appears as:

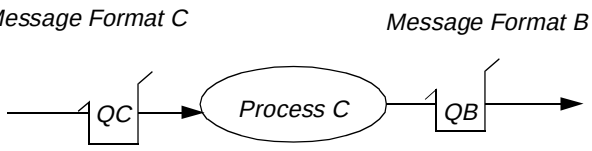


Figure 16. Messaging and queuing

But, if **Process C** is a legacy application which cannot be directly modified to use the MQI, then new **Pre** and **Post** processor applications are required, yielding:

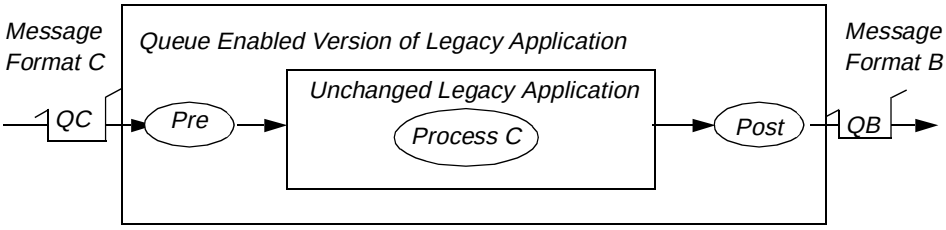


Figure 17. Queue enabled version of legacy application

Of course, the **Pre** and **Post** processors must reside on the same hardware platform as the legacy application.

Planning considerations for UNIX systems

Many of the following are detailed elsewhere in this manual. They are summarized here for convenience.

- *Prerequisite Hardware and Software* is defined in Chapter 2, “Installation” on page 5.
- *MQI Features* have been implemented in a slightly different manner for each operating system environment. Chapter 6, “Application programming interface” on page 107 should be reviewed closely to ensure that any features of particular interest are fully available on UNIX. (In the current release, note especially that *triggering* is not supported under UNIX.)
- *Transport protocols* supported on UNIX include LU 6.2 and TCP/IP. If your enterprise network includes an existing TCP/IP portion, then you may wish to use that as the transport wherever UNIX-to-UNIX logical links are required. To merge TCP/IP with an SNA network, only one UNIX system need be configured to support both protocols. In this configuration, MQSeries System messages may be required to “multi-hop” through the gateway machine. This capability is supported but will have performance implications.
- *The Application Programming Language fully supported on UNIX is C*. If other programming languages are to be used for applications development, the customer must recognize:
  - Only applications written in C have been tested on IBM MQSeries for UnixWare.
  - Source samples are provided in C only.
  - Other languages may be usable at the application level, provided the customer constructs the API interface calls correctly.
- *Security* for MQSeries System queues, communication channels, and administrative programs can be established by taking advantage of native UNIX security features.
- IBM MQSeries for UnixWare is not safe for multi- threaded use and does not support multi-threaded application programs.

Chapter 4. Configuration

The MQSeries System software has now been installed on your system by following the instructions in Chapter 2, “Installation” on page 5. However, it cannot converse with other MQSeries System installations or even perform local messaging until it is *configured*.

Building an effective configuration for the MQSeries System operation is by far the most critical task to insure a successful implementation.

This chapter will explain the *concepts* required to properly configure the MQSeries System.

The mechanics of entering configuration data are detailed in Chapter 5, “System operation” on page 59.

MQSeries System configuration elements

Configuring the MQSeries System requires that the administrator/operator define the following MQSeries System elements:

- Message Queue Manager
- Local Queues
- Dead Letter Queue
- Transmission Queues
- Communications Channels
- Queue Aliases and/or Remote Queue Definitions

A clear understanding of each of these elements and an understanding of MQSeries System message routing is needed in order to properly configure the system.

Queue names and message routing

Queue names are used in all MQI commands to identify the queue with which you want to work. Queue names are also included in message headers and, as the message traverses the network, are the basis for *routing* the message.

A fully qualified queue name consists of two parts:

- The *queue_manager_name*, which identifies an MQSeries System
- The *queue_name*, which identifies the queue itself

The full name may be written *queue_name @ queue_manager_name*.

This two-part naming convention represents the essence of message routing for the MQSeries System. The fundamental routing algorithm is very simple:

> **The *queue_manager_name* identifies the MQSeries System on which the queue called *queue_name* resides.**

If a *queue_manager_name* is not specified, then it is assumed to be the local queue manager (that is, the queue called *queue_name* resides on the local system).

A goal of IBM MQSeries is to hide the network details from the application. The application should not have to identify the system on which a particular queue resides. For this reason, applications can use aliases and remote queue definitions, which are explained later in this section.

Queue name format

Each part of the queue name is contained in a 48-character field (constants `MQ_Q_NAME_LENGTH` and `MQ_Q_MGR_NAME_LENGTH` can be used for the length), with the local-name part appearing first.

The character set that can be used for the local queue manager and local name is as follows:

- Uppercase A - Z
- Lowercase a - z
- Numerics 0 - 9
- Period (.)
- Forward slash (/)
- Underscore (_)
- Percent sign (%)

Note: Leading or embedded blanks are not allowed.

Local and queue manager names that are shorter than the full field width can be passed by an application program, either by padding to the right with blanks, or by using a null (X“00”) character after the last significant character of the name.

The null character and any characters to the right of it (which are ignored), are treated as blanks. There can be blanks between the last significant character of the name and the null character.

For example, a single null character in the first character position of the queue manager name field can be used to default to the connected queue manager. This method is convenient for C programs.

Either method (right padding or null character) can be used for names that are passed by the application across the interface, but all names that are returned by the queue manager are always padded to the right with blanks.

Any structure to the names (for example, the use of the period or underscore) is not significant to the queue manager.

Note: Names starting “SYSTEM” are reserved for the queue manager-defined queues.

Message queue manager

In the distributed LAN architecture, it is most reasonable to think of the Message Queue Manager as the *domain* composed of:

1. The MQSeries System disk directory containing the messaging and queuing configuration database.
2. Any Messaging and Queuing software which operates by using this database.

The latter includes the Message Channel Agent (MCA), the *MQM* operator interface screens, and any MQI applications.

Note: It is possible to have more than one such domain on the same physical LAN. In this case, each instance is a separate Message Queue Manager and must be configured independently. This is illustrated in the following figure.

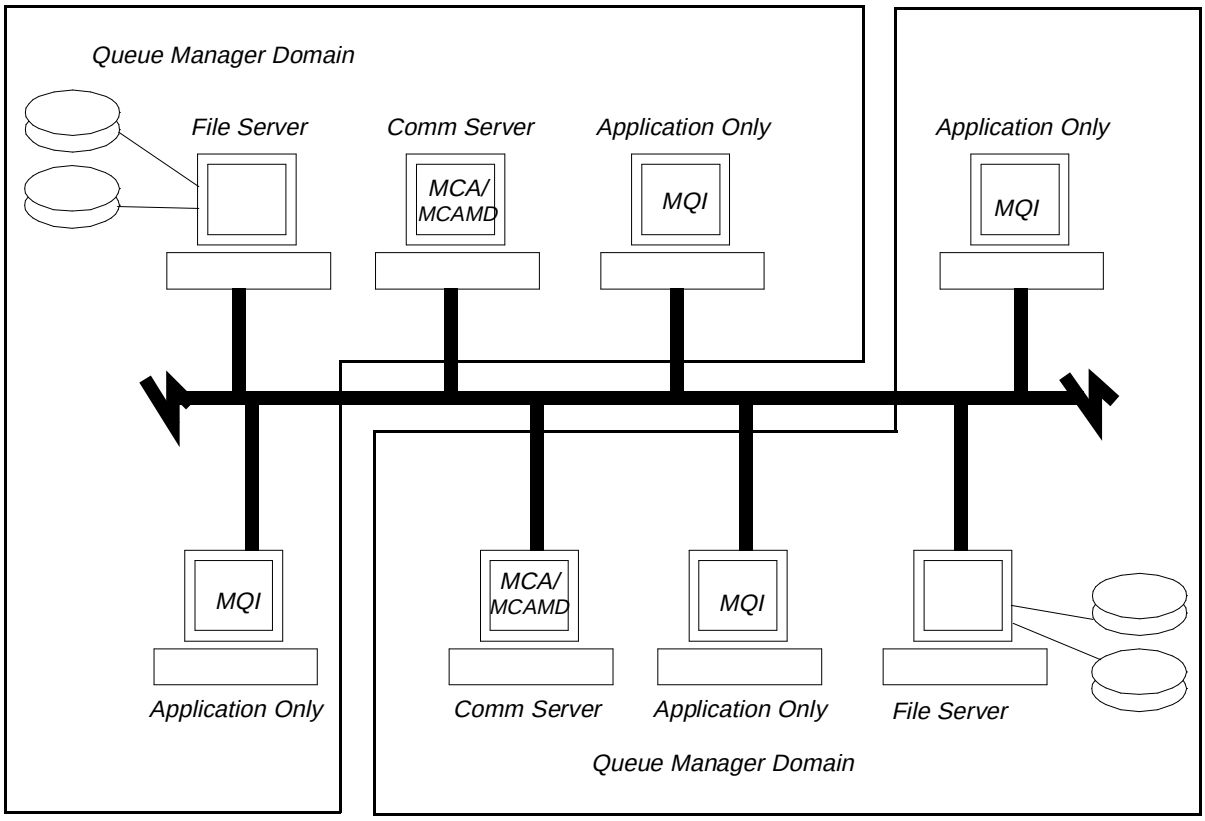


Figure 18. Multiple message queue managers on the same LAN

Local message queues

In Chapter 1, “Product description” on page 1, a *local queue* was defined as any queue residing on the same message queuing system as the application. A local queue corresponds directly to a physical disk file which holds messages.

The *queue_name* of a local queue is used by all programs to access the queue.

In most cases, one local queue must be created for each MQI application running on the system. This queue is used by the MQSeries System to store *inbound* messages destined for the target application. Put another way, the application *receives* messages via its associated local queue.

The required local queues are normally identified by the system designer who has enterprise-wide responsibility for distributed applications. The designer typically associates a *queue name* with an application program. For example, the designer may prescribe the following relationships:

Table 2. Application and queue name

Application	Queue Name
Accounts Receivable	Accts_Receivable
Accounts Payable	Accts_Payable
Order Entry	Ops_Orders
Shipping	Ops_Shipping
Inventory	Ops_Inventory

The publication of a list, as above, establishes a naming convention by which all developers understand how to address messages to a particular destination application. For example, from the above list, any program wishing to send a message to the Order Entry application, uses the MQI command set to *put* a message to the queue named Ops_Orders. Similarly, the Order Entry application itself receives messages (sent by other programs) by using the MQI command set to *get* messages from the queue Ops_Orders.

Note: While the normal case is one local (input) queue per application, there are cases in which an application may require:

- **Multiple local queues** (for example, high priority traffic on a separate queue)
- **No local queues** (for example, programs that generate messages, but never receive)
- **A shared local queue** (for example, multiple processes all servicing the same high volume queue)

Dead letter queue

The dead letter queue is a local queue created expressly for the purpose of redirecting misguided messages. Although this is a simple statement, it is the essence of this queue's purpose.

This queue is created automatically when the queue manager is created. On the Queue Manager modification screen is the field "Dead Letter Q," which will contain the queue name associated with the dead letter queue. A default file name is SYS_DLQ.que in the same directory as all other queues. The message size associated with this queue shall be the Max Message size defined for the Queue Manager. This value may be larger than desired since all messages queued occupy the same number of bytes defined as the queue maximum. Therefore, if the Queue Manager Maximum Message size is 1024 bytes, and a 10 byte message is redirected to the dead letter queue, this message will occupy 1024 bytes on the disk. As a result, a small number of messages have the potential of using a lot of disk space.

Should this situation be seen as a serious problem, the remedy is to delete the dead letter queue that was automatically created and create a new one, using the Create Local Queue configuration screen. You may define a different dead letter queue name, file name, and message maximum size. However, the queue name should agree with that displayed on the Queue Manager screen. If not, modify the Queue Manager record.

Note: It is extremely important that the dead letter queue be immediately recreated should it be deleted for purposes of resizing. It is also important that the queue name be the same as the one specified in the Queue Manager configuration. If the queue does not exist, or should its name not match the one specified in the Queue Manager, the MCA process will exit immediately upon startup.

Applications which put messages directly on the dead letter queue should prefix the message data with an MQDLH structure, and initialize the fields with appropriate values. However, the queue manager does not check that an MQDLH structure is present, or that valid values have been specified for the fields. If a message is too long to put on the dead letter queue, the application message data should be truncated to fit. An `MCA`, however will never truncate the application message.

The MCA process will use the dead letter queue under the following circumstances:

Requester/Receiver Channels

- The remote queue is full.
- The remote queue is PUT inhibited.
- The Message sent to the remote queue is too large.
- The remote queue does not exist.
- The message contains a duplicate Message Sequence Number.

Sender/Server Channels

- The remote Queue Manager places the message it received on its dead letter queue.
- The message on the transmission queue is greater in size than the negotiated Maximum Message Size of the channel.

Note: No feedback codes are generated by the queue manager.

Remote queue definitions

Both a *remote queue definition* and an *alias* are simply an alternative logical name which can be used to address a message queue instead of using the actual *queue_name*. In the case of the remote queue definition, a single name is provided for use by an application which relieves the application of needing to know the location (*queue_manager_name*) of the destination queue.

These extensions to the use of direct *queue_names* exist solely to simplify the work of developers and to improve the flexibility/portability of distributed applications.

A remote queue definition is simply a logical name defined on the local system which identifies a queue physically resident on another system. The *queue_name* so defined, can be used by applications to address the queue, but the MQSeries System will realize the queue is elsewhere and direct the messages to the remote site.

Note: MQSeries does not support GET operations directed to a remote queue.

To define a remote queue, one does not supply the same fields as when defining a local queue (for example, no file name, or record size), but must supply both:

- the *queue_manager_name* of the remote system

and

- the *queue_name* of the actual physical queue on the remote system

Optionally, you may also identify a transmission queue (other than the default transmission queue) which is to be used to send messages to the remote system.

When defining a remote queue, each entered name is validated by the MQSeries System as follows:

queue_name	must be unique among all names defined locally.
queue_manager_name	must match a local transmit queue, or the optional alternative transmit queue must be supplied.
remote_queue_name	must match a definition on the remote system but this cannot be validated locally.
transmit_queue_name	must match a local transmit queue, if present.

This extended queue identity is not visible to an application on the local system. Local applications use only the *queue_name.*

Conversely, the *queue_name* used locally is not visible to the remote system. In its place, the fully qualified remote queue name (*remote_queue_name @ queue_manager_name*) is inserted in the message header before transmission.

Aliases

An *alias* is similar to a *remote queue definition* in that it is an alternative logical name which can be used to address an MQSeries System queue instead of using the actual *queue_name*. An alias, however, is simpler than a remote queue definition.

An alias provides a simple one-to-one name substitution capability. It associates an alternative (alias) name with an already defined queue.

By defining an alias, the MQSeries System administrator has the ability to redirect message traffic. For example, if an application was originally coded to write to a queue called FRED, but we now want the output to go to JOHN, the redirection can be accomplished by redefining FRED as an alias for JOHN rather than as a real local queue.

The MQSeries System supports two other types of aliases beyond the simple queue alias. There is a *manager_alias*, which is simply an alias associated with an already defined queue_manager_name. There is a reply_to_alias which is somewhat more complex and infrequently used. This last type of alias will be explained more fully in “Alias queues, remote queues, and routing” on page 40.

Note: An alias can be defined for a local queue, a remote queue, or a queue manager, but cannot be defined for another alias.

Transmission queues

A special case of a local queue which is used to hold messages to be transmitted to another system is called a *transmission queue*.

Since a transmission queue is a local queue, it also corresponds directly to a physical disk file which holds messages. Beyond that, a transmission queue is substantially different from a *normal* local queue.

Whereas a normal local queue holds inbound messages, a transmission queue holds *outbound* messages. Whereas a local queue holds messages for a single application, a transmission queue interleaves messages destined for several different applications residing on the same remote MQSeries system.

Note: A transmission queue is associated with a communications channel. The messages on a transmission queue are processed by MQSeries System’s Message Channel Agent (MCA). Normal MQI applications cannot directly access a transmission queue for output purposes.

In most cases, one transmission queue must be created for each *adjacent* MQSeries System in the network. In this context, *adjacent* means any system with which you have a point-to-point *logical* connection at the MQSeries System level. Since the physical topology of the underlying transport network is hidden by the MQSeries System, this may or may not correspond to a point-to-point physical connection. But, it may be conceptually easier to think of the connection in physical terms.

While the normal case is one transmission (output) queue per adjacent MQSeries system, there are cases in which an MQSeries System may require:

- **Multiple transmission queues to the same destination** (for example, for higher throughput)
- **No transmission queues** (for example, if the MQSeries System is to be used for only local interprocess communications. While rare in normal operation, this arrangement is often useful in test scenarios.)

Though the system designer typically identifies required local queues, the designer may not identify all required transmission queues. If not, the MQSeries System administrator must compile:

- A list of applications running on the local system
- A list of destination queues to which the local applications send messages
- A list of remote MQSeries systems on which these queues reside

From the above compilation, it should be simple to determine the required transmission queues. Further system or application information will be required to identify special cases requiring more than one transmission queue per connection.

Communications channels

In Chapter 1, “Product description” on page 1, a channel was defined as a unidirectional point-to-point communications link between two MQSeries systems. The MQSeries System channel parameters are defined by using the Channel Definition screen in the *MQM* program, as detailed in Chapter 5, “System operation” on page 59. Each channel has a number of characteristics:

- Twenty character channel name unique to the queue manager
- Message sequence numbers
- Communications parameters required by the transport layer

The MQM Channel Definition screen defines only the communications link parameters. The associated transmission queue must be defined separately.

IBM MQSeries on UnixWare is capable of utilizing both the Common Programming Interface for Communications (CPI-C) API for LU 6.2 and the Transmission Control Protocol/Internet Protocol (TCP/IP).

To configure either one of these transport protocols requires inherent knowledge of the protocol itself. This knowledge may include; the current status of your system, the policies adopted by your network administrator(s), and the procedures to configure these protocols on other systems as well. We will describe the information specifically required for the MQSeries System.

MQSeries utilizing Apertus/SSI’s EXPRESS SNA server

While the menus used by the operator configuration utility (that is, *MQM*) require a limited number of items to be specified, additional configuration is mandatory to establish a working SNA connection. These additional levels are carried out via the EXPRESS SNA configuration control. For information on EXPRESS SNA, please refer to *Systems Strategies EXPRESS SNA Server Configuration Guide* (Release 2.04).

The transport level protocol in this case is implemented as a CPI-C (Common Programming Interface for Communications) Application Program Interface (API). CPI-C provides a consistent programming interface for applications that require program-to-program communication and it makes use of EXPRESS SNA’s LU 6.2 to create a rich set of inter-program services. For the program to run over SNA, certain parameters must be defined. They are, but not limited to:

Local LU Name — A name specifying the local LU (limited to 17 characters) that may either be fully-qualified or locally known.

Partner LU Name — A name specifying the LU on the remote system (limited to 17 characters) that may either be fully-qualified or locally known.

Mode Name — The name of the mode (limited to 8 characters) used by LU 6.2 to designate the properties for the session that will be allocated for the conversation.

TP Name — The Transaction Program (TP) name which represents the partner program on the remote system (limited to 64 characters).

A convenient facility exists for all CPI-C programs in a database called the “Side Information”. It houses the above mentioned parameters among others for each program. It is accessible in each program through a unique index called a:

Symbolic Destination Name Limited to 8 characters.

The MQSeries System user has the option of either choosing to enter the symbolic destination name index or manually entering each parameter.

The SENDER originates data and initiates the remote RECEIVER. The RECEIVER receives the data, initiated by the SENDER. The REQUESTER receives data and initiates the remote SERVER. The SERVER originates data, initiated by the REQUESTER.

The SENDER/REQUESTER channels initiate a conversation between themselves and the RECEIVER/SERVER channel components. The RECEIVER/SERVER channels perform an LU 6.2 Accept Incoming. Through the Accept Incoming, the `MCA` process notifies the SNA controller that it is prepared to maintain a conversation with a transaction program (another `MCA` process) on the remote system.

The remote system actually starts the conversation by performing an LU 6.2 Allocate. When the Allocate arrives at the SNA controller (which has an Accept Incoming outstanding) and the transaction program name specified in both the Accept and Allocate match exactly, the SNA controller notifies the `MCA` process that performed the Accept Incoming, and a conversation is established.

To establish the conversation, it is desirable to start the `MCA` processes that support the RECEIVER/SERVER channels *prior* to starting instances of `MCA` that support SENDER/REQUESTERs. This requirement is not absolute. If the SENDER/REQUESTER is started prior to its remote complementary component, this channel will attempt a retry if configured to do so.

Stopping and starting EXPRESS SNA

EXPRESS SNA must be started, and sessions activated, before starting any `MCA` which will run SNA channels. Similarly, any `MCA` which runs SNA channels must be terminated or killed before stopping EXPRESS SNA.

MQSeries System utilizing TCP/IP

The MQSeries System is capable of using TCP/IP as a transport protocol. The RECEIVER and SERVER channels will open a listener socket accepting one connection from a SENDER or REQUESTER, respectively.

Configuring a channel for TCP/IP requires at most two parameters, a remote hostname (SENDER/REQUESTER only) and a service name.

The remote hostname specifies the system running the complementary channel (RECEIVER/SERVER) on the TCP/IP network. This host must be accessible from the local host over the TCP/IP network. Either the host must be defined in the `/etc/hosts` file or it must be defined to a name server used by the local host. If the host resides in a different network, TCP/IP routing must be set up to allow this access. The service name is an entry you must create in the file `/etc/services` or incorporate into the services portion of the name server database.

The services file provides a mapping between a service name and a port number.

The port number entered:

- should be greater than 1024
- must be identical on both machines
- must be unique within `/etc/services` file or name server

If you are configuring your system within a network administered by network administrators, you should consult them to avoid conflicts in assignment of port numbers.

The following is an example of an entry in the file `/etc/services`:

Table 3. etc/services format

`channel1`	`3000/tcp`	`# Channel`
`channel2`	`3001/tcp`	`# Channel`

A port number can service:

- one or more SENDER channels, and
- one or more REQUESTER channels, and
- one RECEIVER or SERVER (not both) channels

at the same time. For example, port number 1414, with service name `MQSERIES` could be used by all local sender and requester channels that connect to remote V2 MQSeries systems and by exactly one local receiver or server channel that receives connections from a remote MQSeries system.

To configure multiple RECEIVER and SERVER channels active at the same time, the `/etc/services` file must contain a different service name and port number for each RECEIVER and SERVER.

The recommended configuration technique is as follows:

- Use only port number 1414 for SENDER and REQUESTER channels to talk to Version 2 IBM Message Queue Managers. Give the service a name such as `V2MQSERIES` or `MQSERIES`.
- Allocate a port number to each RECEIVER or SERVER channel. The SENDER or REQUESTER at the other end of the channel must use this same port number. A different port number must be used for each channel pair.
- Allocate the same port number to each SENDER or REQUESTER channel as used by the RECEIVER or SERVER at the other end of the channel. A different port number must be used for each channel pair.
- Name the service for the port the same as the channel name (except for those channels using the "`V2MQSERIES`" service). This will avoid confusion.

Port number 1414 is assigned by the Internet Assigned Numbers Authority to MQSeries.

Example: To set up SENDER and RECEIVER channels from T1 to M1, M2, T2, and T3, the channels can be configured as follows at T1 (T1, T2, and T3 are Version 1 IBM MQSeries for UnixWare Queue Managers, and M1 and M2 are Version 2 IBM Message Queue Managers):

Table 4. etc/services example

Channel name	Channel type	Port number	Service name
CHANNEL_T1_TO_M1	SENDER	1414 (listener at M1)	V2MQSERIES
CHANNEL_T1_TO_M2	SENDER	1414 (listener at M2)	V2MQSERIES
CHANNEL_T1_TO_T2	SENDER	3000 (1st listener at T2)	CHANNEL_T1_TO_T2
CHANNEL_T1_TO_T3	SENDER	3004 (5th listener at T3)	CHANNEL_T1_TO_T3
CHANNEL_M1_TO_T1	RECEIVER	3000 (1st listener at T1)	CHANNEL_M1_TO_T1
CHANNEL_M2_TO_T1	RECEIVER	3001 (2nd listener at T1)	CHANNEL_M2_TO_T1
CHANNEL_T2_TO_T1	RECEIVER	3002 (3rd listener at T1)	CHANNEL_T2_TO_T1
CHANNEL_T3_TO_T1	RECEIVER	3003 (4th listener at T1)	CHANNEL_T3_TO_T1

Note: If the Internet Protocol is shut down on a remote system that is supporting TCP/IP-based channels, the local channels will not receive notification of the shutdown. Therefore, the local channels will remain in the state they were in at the time of the shutdown.

MQSeries System channel implementation

To fully implement an MQSeries System channel, you must perform several functions on each of two systems. For example, in order to connect SystemA (running IBM MQSeries for UnixWare) to SystemB (running the same or another MQSeries system), you must:

On the MQSeries System system installed on SystemA:

- Use the *MQM* Queue Definition screen to define an outbound transmission queue called, for example, **QUE_TO_SYS_B**.
- Use the *MQM* Channel Definition screen to define communications parameters for the channel named, for example, **CHANNEL_1**. In the channel definition, you will specifically identify **QUE_TO_SYS_B** as the name of the transmission queue for this channel.
- Insure all transport layer hardware and software is properly installed.
- Activate the `MCAMD` and `MCA` to process the SystemA end of **CHANNEL_1**.

On the MQSeries system installed on SystemB:

- Insure all transport layer hardware and software is properly installed.
- Use the MQSeries System channel definition function to define communications parameters for the channel named **CHANNEL_1**. (On the receiving end of a channel, no transmission queue is involved.)
- Activate the `MCA`, supplied with the MQSeries system, to process the SystemB end of **CHANNEL_1**.

Note: The above sequence of actions has established a channel in one direction only, from SystemA to SystemB. The same steps must be performed to create another channel, if desired, to allow messages to flow in the opposite direction.

MQSeries System message routing

MQSeries System message routing is not to be confused with lower level network routing. The MQSeries System is normally concerned only with fixed, point-to-point routing which is substantially simpler than dynamic, adaptive, multi-hop, network routing algorithms. However, the many options available can make MQSeries System routing somewhat complex.

The MQSeries System must be explicitly configured (that is, queues, channels, aliases, etc. must be defined) to insure the desired flow of messages through the network. To do this effectively, the MQSeries System routing algorithm must be understood.

To understand MQSeries System routing, we will look first at the basic routing algorithm, then at the MQSeries System routing table, and finally at effects on routing which can be generated through aliases and remote queue definitions.

Basic message routing

Early in this chapter, it was noted that the two-part MQSeries System queue names embodied the essence of message routing for the MQSeries System. The fundamental routing algorithm is very simple:

> **The *queue_manager_name* identifies the MQSeries System on which the queue called *queue_name* resides**

The basic algorithm may be expanded by following the flow of a typical message from one system to another, as follows:

1. At the originating system, a message is presented (PUT) to the MQI with the two-part destination *queue_name*.
2. The MQSeries System examines the destination *queue_manager_name* to see if it matches the *local_queue_manager_name*. Typically, it does not match, so the MQSeries System knows the message goes to another system.
3. In this case, the destination *queue_manager_name* **must** match a transmission queue defined on the originating system. This is the default transmission queue to reach the specified *queue_manager.*
4. The message is enqueued to this transmission queue.
5. The MQSeries System MCA on the originating (output) system GETs the message from the transmission queue and sends it over the link to the remote system. Notice that the output MCA utilizes no routing logic.
6. The MCA on the destination (input) MQSeries system receives the message from the communications link (and invokes routing logic to determine what to do with it).
7. The MQSeries System examines the destination *queue_manager_name* to see if it matches the *local_queue_manager_name*. Typically, it does, so the MQSeries System knows the message belongs “here.”
8. The MQSeries System then examines the destination *queue_name.* In this case, the destination *queue_name* **must** match a local queue defined on the destination system.
9. The message is enqueued to this destination queue.
10. The destination application receives (GETs) the message via the MQI.

From this example, several **basic configuration principles** may be observed:

1. MQSeries System routing logic is exercised independently on each system. Therefore, each MQSeries System must be configured individually, but all configurations must be coordinated to be effective.
2. Any configuration must have defined one local queue for each inbound destination on the system. These will be used to receive incoming messages. The defined *queue_name* must match the *queue_name* applications will use in message headers.

3. Any configuration must have defined one transmission queue for each remote destination system. These will be used to transmit outbound messages. The defined *queue_name* must match the *queue_manager_name* applications will use in headers of outbound message.

The MQSeries System routing table

Before exploring the routing logic further, it is useful to understand the MQSeries System's **Routing Table** which is used to resolve all queue references. Note that the table is described here as a logical entity and may not exactly correspond to the data structure on a particular system.

The MQSeries System queue names are of the form **queue_name @ queue_manager_name**, each half of which is 48 characters. The MQSeries System Routing Table, however, is keyed to a single 48-character string. This string is normally a *queue_name* but will be called *Object_Name* to avoid/reduce confusion in this discussion.

An entry must exist in the Routing Table for each of the following:

- All **LOCAL** queues (Type=Local, Usage=Normal)
- All **TRANSMISSION** queues (Type=Local, Usage=Transmission)
- Any desired definitions for **REMOTE** queues
- Any desired **ALIAS_Q** names for queues
- Any desired **ALIAS_M** names for queue_managers
- Any desired **ALIAS_R** names for reply_to_queues

The format of each Routing Table entry varies according to type. This is summarized in the chart below.

Table 5. Routing table format

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
Required	**Local**	----	----	----
Required	**Transmit**	----	----	----
Required	**Remote**	Required	Required	Optional
Required	**Alias_Q**	Required	----	----
Required	**Alias_M**	----	Required	Optional
Required	**Alias_R**	Required	Required	----

Alias queues, remote queues, and routing

What happens to routing, when alias queues and remote queues are introduced into the algorithm? This can be seen by considering various routing scenarios and examining the results of these scenarios under the MQSeries System routing logic. These sample routing cases will also further illustrate normal routing cases.

In these cases, the **QName @ QMgrName** shown as input indicate the “actual” nature of the input. The routing algorithm results will be as indicated for each case.

Note: The following is not intended to suggest application level pseudo-code, but only to explain what happens within MQSeries System routing.

1. Application attempts to operate on queue identified as

   Queue_Name @ Local_Queue_Manager

   Routing Process:

   Queue_Name matches Routing Table entry for Local or Remote queue .... or error.
2. Application attempts to operate on queue identified as

Queue_Name @ (Blank_Queue_Manager)

Routing Process:

Queue_Name matches Routing Table entry for Local or Remote queue .... or error.

3. Application attempts to operate on queue identified as

   Queue_Name @ Remote_Queue_Manager

   Routing Process:

   Remote_Queue_Manager matches Routing Table entry for Transmit queue .... or error, ***Queue_Name*** is ignored.

4. Application attempts to operate on queue identified as

   Remote_Queue_Name @ (Blank_Queue_Manager)

   Routing Process:

   Remote_Queue_Name matches Routing Table entry for Remote queue .... or error.

5. Application attempts to operate on queue identified as

   Alias_Name @ Local_Queue_Manager

   Routing Process:

   Alias_Name matches Routing Table entry which resolves to another Routing Table entry for Local or REMOTE queue .... or error.

6. Application attempts to operate on queue identified as

   Alias_Name @ (Blank_Queue_Manager)

   Routing Process:

   Alias_Name matches Routing Table entry which resolves to another Routing Table entry for Local or REMOTE queue ... or error.

7. Application attempts to operate on queue identified as

   Alias_Name @ Remote_Queue_Manager

   Routing Process:

   Remote_Queue_Manager matches Routing Table entry for Transmit queue ... or error, ***Alias_Name*** is ignored.

8. Application attempts to operate on queue identified as

   Some_Queue_Name @ Alias_Queue_Manager

   Routing Process:

   Alias_Queue_Manager matches Routing Table entry with type ALIAS_M which resolves to ***Local_Queue_Mgr_Name***. Second pass through search logic resolves ***Some_Queue_Name*** to either case (1) or (5) above.

   OR

   Alias_Queue_Manager matches Routing Table entry with type ALIAS_M which does NOT resolve to ***Local_Queue_Mgr_Name.***

   This case is handled same as case (3) or (7) above. No second pass through the search logic is required. ***Some_Queue_Name*** is ignored.

   OR

   Alias_Queue_Manager matches nothing, and is an error.

9. Message Channel Agent receives inbound message with destination queue identified as:

   Some_Queue_Name@Some_Queue_Manager

   Routing Process:

   Some_Queue_Manager matches ***Local_Queue_Manager_Name*** and ***Some_Queue_Name*** is resolved as in case (1) or (5) above with a single pass through the search logic.

OR

Some_Queue_Manager matches Routing Table entry with type

Alias_M which resolves to the ***Local_Queue_Manager_Name***.

On second pass through search logic, ***Some_Queue_Name*** is resolved as in case (1) or (5) above.

OR

Some_Queue_Manager matches Routing Table entry with type

ALIAS_M which does NOT resolve to ***Local_Queue_Mgr_Name.***

This case is handled same as case (3) or (7) above. No second pass through the search logic is required. ***Some_Queue_Name*** is ignored.

OR

Some_Queue_Manager matches Routing Table entry with type Transmit and is handled same as case (3) or (7) above.

Some_Queue_Name is ignored.

OR

Some_Queue_Manager is invalid.

Other alias types

In some cases, it is desirable to have multiple channels, and multiple transmit queues defined for the same remote destination system. This conflicts with the standard use of the queue_manager_name as the transmit queue name.

The extension of the REMOTE queue definition to include a TRANSMIT queue is convenient for most such cases, but may be undesirable at a central “server” system which must deal with a large number of remote systems. The server would require a large number of REMOTE queue definitions in order to handle anything more than one TRANSMIT queue per system.

A Routing Table entry type ALIAS_R provides a mechanism to allow the name for the response transmission queue to be expanded at the originating system.

This may be thought of as a “Reverse Queue Manager Alias” or as a “Response Class” or as a “Response Category”.

It is a relatively simple concept which simultaneously frees a remote server from the need to define a long list of REMOTE queues, and frees the local application from the need to know details of the transmit queue structures, and allows the local application code to be completely portable.

For example:

This example shows the use of reply aliases and manager aliases to reduce the definitions required at a central server site.

An application running on SYS1 originates a message to a remote server and specifies **Reply_to_Queue** = PRIORITY (and ***Reply_to_Queue_Manager*** = BLANK).

At SYS1, the Routing Table contains three related entries:

Table 6. Local routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
My_Queue	LOCAL	----	----	----
PRIORITY	ALIAS_R	My_Queue	SYS1_PRI	----
SYS1_PRI	ALIAS_M	----	SYS1	----

During outbound processing, MQSeries System finds that PRIORITY matches a Routing Table entry of type ALIAS_R, and substitutes MY_QUEUE @ SYS1_PRI into the outbound ***Reply_to_Queue*** fields.

At the remote server, the Routing Table contains one related entry:

Table 7. Remote server routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
SYS1_PRI	Transmit	----	----	----

When the server has completed processing the original message, the response is queued to the transmit queue SYS1_PRI.

Back at SYS1, the response arrives with ***QMgrName***=SYS1_PRI. This is resolved through the Routing Table to match SYS1 and so the message is accepted and enqueued to the local queue MY_QUEUE.

Portable application code:

Not only did the above result in minimizing Routing Table entries at the server, but also it promotes totally portable application code.

Consider the case in which the network in the above example is to be expanded by adding a new system called SYS2. The new system will run the same application software as SYS1, and will post requests to the same server application.

By simply copying the unmodified application (executable) code from SYS1 to SYS2, and making the following Routing Table updates, all will work correctly.

At SYS2, the Routing Table contains three related entries:

Table 8. Additional system routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
My_Queue	LOCAL	----	----	----
PRIORITY	ALIAS_R	My_Queue	SYS2_PRI	----
SYS2_PRI	ALIAS_M	----	SYS2	----

At the remote server, the Routing Table expands by only one related entry:

Table 9. Remote server's new routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
SYS1_PRI	Transmit	----	----	----
SYS2_PRI	Transmit	----	----	----

At both SYS1 and SYS2, the application code uses the name "PRIORITY" as the ***Reply_to_Queue.*** The ALIAS_R logic resolves this correctly via the Routing Table and correctly directs response traffic through the server to two different remote systems through TRANSMIT queues which are not the default queues.

This, of course, can be extended to any number of *Message_Queue_Managers* and to any number of TRANSMIT queues used for Response messages.

Recommended naming conventions

The naming (of queue_managers, queues, and aliases) used in the MQSeries System can be very flexible. Each organization will have its own view of how these names should be constructed. Beyond conforming to the format described in “Queue name format” on page 30, choosing names is left entirely in the hands of the user organization. However, a few suggestions are provided below.

1. **Don't use very long names:** Though the name fields are 48 characters long, very long names are cumbersome. Also, in some cases, the MQSeries System displays or messages may truncate very long names due to screen size limitations. In most cases names substantially shorter than 48 characters are sufficient.
2. **Attempt to configure the MQSeries System so that all queues may be referred to by a one-part name:** This will maximize the “network independence,” or minimize the network topology knowledge required, of the distributed applications. It is desirable for applications to use only a *queue_name* rather than the two-part *queue_name@queue_manager_name* construct, allowing the MQSeries System, through its routing table, to determine the location of the queue. This can easily be accomplished by using remote queue definitions and/or aliases to identify all remote queues. (In installations which require access to a large number of remote queues, this may be too cumbersome to configure.)
3. **Use aliases:** First, this can avoid the need to change application source code when the network changes or when a remote application changes. Also, aliases can be used to resolve incompatibilities between different naming domains. For example, if two computers in different companies are talking via MQSeries Systems, each company will probably want to name their own queues and queue managers. A “territorial” dispute is not uncommon. Such conflicts can be resolved by using aliases to “translate” names at the “border”.

Configuration capacities

In the current release, the major configuration elements are limited as follows:

Queue Managers:	One.
Local Queue Definitions:	Unlimited.
Alias Definitions:	Unlimited.
Remote Queue Definitions:	Unlimited.
Total Queue Objects:	Unlimited.
Object Handles:	Unlimited. (The User may set a maximum number via the Define Queue Manager screen, but this number is currently ignored).
Channels:	Maximum of 20 per MCA process.
MCA Processes:	Maximum of 9.
Maximum Message Size:	64,000 bytes for LU 6.2 and TCP/IP, excluding header information. The User may set a system-wide maximum message size up to 64,000 bytes via the Define Queue Manager screen. The User may also set a maximum message size up to the system-wide maximum message size for each queue via the queue definition screen.

System disk space requirements for the MQSeries System

The following equation may be used to provide an indication of the system disk space requirements. This is a general guide, the user should be aware of any particular needs and restrictions that his own system setup may impose.

```
Total system disk space required =
            FMR MegaBytes + BMQ bytes + QDT bytes + IDX bytes + CDM bytes + LOG bytes
```

where the values FMR, BMQ, QDT, IDX, CDM, LOG are described below.

In the process of determining these values, it is necessary to estimate the number of queues and channels the user will be creating and defining. While impossible to determine an exact number, there are certain assumptions that can be made. A novice user will tend to delete more queues or channels, as they may “experiment” or make more errors. An experienced user will tend to delete very few queues or channels, as they only need to delete queues that were incorrectly created. Therefore, a reasonable assumption is that the number of queues created will generally be approximately twice the number of channels needed. When a queue or channel is deleted, the values of QDT, IDX and CDM are not decremented; system disk space is not freed up when a user deletes a queue, but is reused when a new queue is defined. Since these values are relatively small, equations are given in the following descriptions to provide a reasonable estimation.

To compute the system disk space requirements for the MQSeries System, the following values need to be determined:

1. Fixed system disk space requirements (FMR) :
   Installed files take up 3.2 MegaBytes.
   Therefore FMR = 3.2 MegaBytes.
2. Variable system disk space requirements :
   There are 4 variable system disk space requirements.
   a. The Basic system disk space requirement for each Local Queue (BMQ) :
   The value of BMQ is in bytes.

      The MQSeries System creates two files for each local queue defined. They are xxx.que and xxx.qul, where xxx is the local file name that the user has configured when defining the local queue. The system disk space requirements for the two files can be determined for each local queue created by using the following formula to compute the value of BMQ for each local queue.

      ```
      BMQ = (NumMsgs * (MaxRecordSize + 588 bytes)) + 64 bytes
      ```

      where:
      NumMsgs is the number of messages on a particular queue.
      MaxRecordSize is Max Message value defined for a particular queue.

      The value BMQ represents the actual value of bytes used at a particular time. To determine the maximum value of BMQ, for each local queue, the user will have to estimate the maximum number of messages on the queue.

   b. The Queue Definition Table system disk space requirement (QDT).
   The value of QDT is in bytes and represents the total number of queues defined.

      The MQSeries System creates a file called SYSTEM.qdt. This file contains configuration information for each local, remote and alias queue defined by the user. The system disk space requirements for the file SYSTEM.qdt, can be determined by computing the value of QDT.

      Use the following approximation to determine the value of QDT:

      ```
      QDT = 320 bytes * Total # of Queues Created
      ```

      where:
      Total # of Queues Created is the number of queues the user has created, regardless of how many queues they have deleted.

c. The Index Table system disk space requirement (IDX).
The value of IDX is in bytes and represents the number of local queues defined.

The MQSeries System creates a file called SYSTEM.idx. This file contains additional configuration information for each local queue defined by the user. The system disk space requirements for the file SYSTEM.idx, can be determined by computing the value of IDX.

The number of local queues created is usually approximately twice the number of sender/server channels plus the number of receiver/requester channels.

Use the following approximation to compute the value of IDX:

```
IDX = 56 bytes * Total # of Local Queues Created
```

where:
Total # of Local Queues Created is the number of local queues the user has created, regardless of how many local queues they have deleted.

d. The Channel Database requirement (CDM).
The value of CDM is in bytes and represents the number of channels defined.

The MQSeries System creates a file called SYSTEM.cdb. This file contains configuration information for each channel defined by the user. The system disk space requirements for the file SYSTEM.cdb, can be determined by computing the value of CDM.

Use the following approximation to compute the value of CDM:

```
CDM = 460 bytes x Total # of Channels Configured
```

3. Log file

The log file(s) is always placed in the directory “/var/mqi/log”. It is recommended that the user has at least one MegaByte available in the “/var” partition available for the MQSeries System. To determine the amount of total and available system disk space in the “/var” partition, use the system command “/etc/dfspace” (if available) or “/etc/df”. The MQSeries System will write as much data as is available in the “/var” partition to the log file, and can use up all the system disk space in this partition. When the “/var” partition is filled, all subsequent log messages are written to standard error, and not to the log file. Typically, a user does not need a large amount of system disk space for the log file. A user would only need a large amount of system disk space if any of the following conditions apply:

a. The tracing function is turned on for a channel.
b. The user does not periodically clean up old log files.
c. The user performs an extraordinary large number of `MCA` start-ups and shutdowns.

While one MegaByte of space is recommended, a more typical range of values needed for the log file is two to five MegaBytes.

Call the value needed for the log file LOG.

As an example, consider the following:

A user will need 10 channels:
4 sender channels
3 receiver channels
2 requester channels
1 server channels

The user defines 15 local queues, which is twice the number of sender/server plus the number of receiver/requester channels (15 = (2 * (4 + 1)) + (3 + 2)).

Each local queue has a value of 10 KiloBytes for the Max Message value. The user assumes that there will be a maximum of 500 messages on each queue. Therefore, NumMsgs = 500 and MaxRecordSize = 10 KiloBytes for each queue.

The user also creates 5 remote queues and no alias queues.

Since this is an experienced user and the amount of queues that they delete is low, we can assume that only 2 queues will need to be deleted, due to typographical or configuration error. Therefore, the total number of queues the user created was 22, although 2 queues were deleted.

The total number of local queues created was 15, as the user did not delete any local queues.

The user determines that the log file should be 2 MegaBytes.

Now for the calculations:

The value of FMR = 3.2 MegaBytes.

The value of BMQ = (500 x (10240 + 588)) + 64) = 5414064 bytes or about 5.28 MegaBytes.

The value of QDT = 320 x 22 = 7040 bytes.

The value of IDX = 56 x 15 = 840 bytes.

The value of CDM = 460 x 10 = 4600 bytes.

The value of LOG = 2 MegaBytes.

Total System Disk Space Required = 3.2 MegaBytes + 5.28 MegaBytes + 2 MegaBytes + 7040 bytes + 840 bytes + 4600 bytes = 10751344 bytes or about 10.5 MegaBytes.

Notice that in this example the values of QDT, IDX and CDM are usually small compared to the values of FMR, BMQ and LOG. This may not always be the case, depending on how many queues and channels the users defines.

Configuration worksheets

The set of sample worksheets is provided in Appendix D, “Configuration worksheets” on page 189, and is presented in a format intended for duplication and use by the MQSeries Systems administrator or other individuals who design, configure, or require knowledge of the MQSeries System network.

The worksheets presented are:

- System List (Message Queue Manager Names)
- Application List (Queue Names and Host Systems)
- Application Look at Queues
- System Look at Queues
- Channel List
- MQSeries System Configuration (Routing Table) Work Sheet

Each of the worksheets is presented one-worksheet-per-page on the following pages. The purpose and field descriptions appear at the beginning of each worksheet. Users may use all, some, or none of these worksheets at their discretion.

Configuration examples

Four sample configurations are presented below.

Simple network - minimum configuration

Consider two systems, one in Chicago, one in Boston. Each system has a single Message_Queue_Manager which has the same name as the host city.

Both Chicago and Boston run copies of the same two applications, **Application_1** and **Application_2**, which are served by local queues **App_1** and **App_2** respectively.

Any application must be able to talk to any other application, but no segregation of traffic is required on the transmission between nodes. So, the default transmission queues are sufficient.

Table 10. Minimal Boston routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
App_1	Local	---	---	---
App_2	Local	---	---	---
Chicago	Transmit	---	---	---

Table 11. Minimal Chicago routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
App_1	Local	---	---	---
App_2	Local	---	---	---
Boston	Transmit	---	---	---

With the above configuration, applications at Boston may put messages to:

App_1 (The LOCAL application)

or

App_1 @ **Chicago** (The REMOTE application)

or

App_2 (The LOCAL application)

or

App_2 @ **Chicago** (The REMOTE application)

Similarly, applications at Chicago may put messages to:

App_1 (The LOCAL application)

or

App_1 @ **Boston** (The REMOTE application)

or

App_2 (The LOCAL application)

or

App_2 @ **Boston** (The REMOTE application)

Simple network - improved configuration

This simple configuration in the preceding example is workable, but it requires the applications to be aware of “Boston” and “Chicago” as the existing transmission queues.

This configuration could be improved as follows:

Table 12. Improved Boston routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
App_1	Local	---	---	---
App_2	Local	---	---	---
Rem_App_1	Remote	App_1	Chicago	---
Rem_App_2	Remote	App_2	Chicago	---
Chicago	Transmit	---	---	---

With the above configuration, and complementary changes to the Chicago Routing Table, applications at either Chicago or Boston may put messages to:

App_1 (The LOCAL application)

or

Rem_App_1 (The REMOTE application)

or

App_2 (The LOCAL application)

or

Rem_App_2 (The REMOTE application)

Simple network - improved configuration #2

A similar result could also be achieved with an alternative Routing Table using ALIAS_M entries, for example:

Table 13. Improved Boston routing table using ALIAS_M

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
App_1	Local	---	---	---
App_2	Local	---	---	---
Remote	Alias_M	---	Chicago	---
Chicago	Transmit	---	---	---

With the above configuration, and similar changes to the Chicago Routing Table, applications at either Chicago or Boston may put messages to:

App_1 (The LOCAL application)

or

App_1 @ Remote (The REMOTE application)

or

App_2 (The LOCAL application)

or

App_2 @ Remote (The REMOTE application)

Complex network - recommended configuration

Consider three host systems, one in Chicago, one in NewYork, one in Boston. Each of these systems has a single Message_Queue_Manager which has the same name as the host city.

Both Chicago and Boston run copies of the same four applications, **Application_1**, **Application_2**, **Application_3**, and **Security**. At both locations, these applications are served by local queues **App_1**, **App_2**, **App_3,** and **Sec** respectively. The first three applications at these sites interact only with a server at New York but not with each other. Additionally, **App_3** uses a segregated priority transmission queue.

New York is a centralized server site running two applications, **Server** and **Security. Server** is an “advanced” application which is served by two local queues **Nor_Req** and **Pri_Req**. Typically the remote applications **#1** and **#2** send normal traffic to **Nor_Req**. Application **#3** sends “high priority requests” to **Pri_Req**.

At all three locations, the **Security** applications may talk to any other **Security** application but their “classified” traffic must be segregated from the other applications' traffic. That is they must have a separate transmission queue.

Finally, in addition to these 3 host systems, there are fifty (50) distributed LANs, one in every state. Each LAN supports up to 20 applications which can generate both normal and priority requests to **Server** at New York. The normal and priority traffic must have segregated transmission queues to and from the server system.

Table 14. Boston host routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
App_1	Local	---	---	---
App_2	Local	---	---	---
App_3	Local	---	---	---
Sec	Local	---	---	---
NewYork	Transmit	---	---	---
NY_Priority	Transmit	---	---	---
NY_Secure	Transmit	---	---	---
Chicago	Transmit	---	---	---
Chi_Secure	Transmit	---	---	---
Sec_NY	Remote	Security	NewYork	NY_Secure
Sec_Chi	Remote	Security	Chicago	Chi_Secure
Nor_Req	Remote	Nor_Req	NewYork	---
Pri_Req	Remote	Pri_Req	NewYork	NY_Priority
Pri_Reply	Alias_R	App_3	Boston_Pri	---
Boston_Pri	Alias_M	---	Boston	---

Table 15. Chicago host routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
App_1	Local	---	---	---
App_2	Local	---	---	---
App_3	Local	---	---	---
Sec	Local	---	---	---
NewYork	Transmit	---	---	---
NY_Priority	Transmit	---	---	---
NY_Secure	Transmit	---	---	---
Boston	Transmit	---	---	---
Bos_Secure	Transmit	---	---	---
Sec_NY	Remote	Security	NewYork	NY_Secure
Sec_Bos	Remote	Security	Boston	Bos_Secure
Nor_Req	Remote	Nor_Req	NewYork	---
Pri_Req	Remote	Pri_Req	NewYork	NY_Priority
Pri_Reply	Alias_R	App_3	Chicago_Pri	---
Chicago_Pri	Alias_M	---	Chicago	---

Table 16. New York host routing table

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
Nor_Req	Local	---	---	---
Pri_Req	Local	---	---	---
Sec	Local	---	---	---
Chicago	Transmit	---	---	---
Chicago_Pri	Transmit	---	---	---
Chicago_Sec	Transmit	---	---	---
Boston	Transmit	---	---	---
Boston_Pri	Transmit	---	---	---
Boston_Sec	Transmit	---	---	---
Alabama	Transmit	---	---	---
Alabama_Pri	Transmit	---	---	---
Repeat above pair for all 50 State LANs				
Wyoming	Transmit	---	---	---
Wyoming_Pri	Transmit	---	---	---
NY_Priority	Alias_M	---	NewYork	---

Table 17. State LAN routing table (identical at each site except for state name)

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
LAN_App_1	Local	---	---	---
Repeat above for all LAN Applications				
LAN_App_20	Local	---	---	---
NewYork	Transmit	---	---	---
NY_Priority	Transmit	---	---	---
Nor_Req	Remote	Nor_Req	NewYork	---
Pri_Req	Remote	Pri_Req	NewYork	NY_Priority
Nor_Reply_1	Alias_R	LAN_App_1	*StateName*	---
Pri_Reply_1	Alias_R	LAN_App_1	*StateName*_Pri	---
Repeat above two lines for all LAN Applications				
*StateName*_Pri	Alias_M	*StateName*	---	---

With the above Routing Table configurations, applications at either Boston or Chicago or any of the 50 State LANs may PUT messages to:

Nor_Req (to **Server** at **NewYork** via “normal” path)

or

Pri_Req (to **Server** at **NewYork** via “fast” path including segregated Transmission Queue)

Any replies from **Server** may be specified by the originating application to be returned via **Pri_Reply** (or **Pri_Reply_n** for LAN Applications) which will use the segregated transmission queue for high-priority responses from New York back to whichever system originated the request.

Also, applications at the Boston or Chicago hosts may PUT messages to:

Sec_NY (to **Security** at **NewYork** via “secure” path including segregated Transmission Queue)

Notice that the **Security** applications are fully defined as REMOTE queues at all of the three major hosts. Thus no ALIAS_R routing entries are required, yet all traffic (including responses) can flow over the segregated Secure transmission queues.

Finally, consider the Routing Table entries required to support the **Server** application at NewYork. Entries are needed for 106 TRANSMIT queues (2 to each of 50 LANs and 3 to each of the other hosts). While this is a large number of entries, realize that it allows for segregated responses to each of more than 1,000 applications (20 at each LAN plus those at the hosts). To provide this same capability using only REMOTE queue definitions at the server (and not using ALIAS_R logic) would require the New York Routing Table to be over 2,000 entries (a normal path and a priority path to each remote application).

IBM MQSeries Version 1 UNIX product configuration guidelines

Channel configuration guidelines

When configuring the channel, use the following guidelines for the following fields:

Disconnect Timeout

The Disconnect Timeout field represents the number of seconds before a sender or server channel disconnects due to an idle (empty) transmit queue.

For SNA Channels, consider the LU6.2 resources being used by both ends of the channel. For example, on a mainframe using CICS, it is relatively “expensive” in terms of system resources, to keep a LU 6.2 transaction up and active (for example, SNA sessions per LU may be limited). Therefore, typically the Disconnect Timeout on a mainframe, will be very small (in the order of 1 to 5 seconds).

On most UNIX and Tandem operating systems, it is relatively “inexpensive”, in terms of system resources, to keep a transaction up and active. Therefore, either a large timeout value (in the order of 60 to 300 seconds) or even the infinite value of “0” may be acceptable.

For TCP/IP Channels, the only major factor to consider is the actual setup time for the TCP/IP connections to complete. If the setup time is large, you may want to consider keeping the Disconnect Timeout value large (in the order of 60 to 300 seconds) or infinite. If the setup time is small, then the Disconnect Timeout value can be kept to a relatively small value (in the order of 5 to 30 seconds).

The default value for the Disconnect Timeout when creating a channel definition is “0”. With this value, the sender or server channel will never disconnect normally except by a channel disable command.

Retry Count

The Retry Count field represents the number of times a connection is retried either when communications have not been established, or when communications have been established but have subsequently disconnected abnormally. Each retry is performed at the interval specified in the Reconnect Timeout field.

If the Retry Count is too small and the channel is constantly disconnecting, then the channel will exhaust its retries, the channel status on the Monitor Channel screen will show up as DISABLED, and operator intervention will be necessary to enable the channels.

If the Retry Count is large, then connection retries will proceed automatically without operator intervention. This is usually the preferred mode of operation. However, the Monitor Channel status for such a channel will show a status of IDLE and will not appear to be having a problem. Only by monitoring the queues and noting that the channel's transmit queue depth is non zero and not decreasing, or by monitoring the log file and noting the repetitive connect attempts and failures, will the operator realize there is a channel problem. The other drawback to a large Retry Count, is that it may use a significant amount of some system resources, especially if SNA channels are being used.

The default value for the Retry Count when creating a channel definition is “0”. With this value, connection retries will not occur. It is usually better to have a large Retry Count specified, in the order of 100 to 10000.

Reconnect Timer

The Reconnect Timer field represents the length of time in seconds before a reconnect is attempted.

The only consideration is not to make the Reconnect Timer too small of a value, since it is possible to retry a connection too quickly. In this case, the retry will most likely fail, and system resources will be wasted. Therefore, a value larger than 15 seconds is recommended.

The default value for the Reconnect Timer when creating a channel definition is “0”.

Checkpoint Count

The Checkpoint Count field represents the number of messages that will be sent or received before a checkpoint of the channel information will be written to the system channel database.

If this value is low, in the order of 1 to 5, there is more system overhead I/O being performed, reducing the channel throughput (messages per second), and also reducing the channel restart time (message sequence number recovery) after an MCA restart.

If this value is high, in the order of 50 or greater, then channel throughput will be higher, but channel recovery will take longer.

The default value chosen for the Checkpoint Count is “10”, as this represents a good compromise, between system overhead and channel recovery overhead.

Line Check Timeout

The Line Check Timeout is the length of time in seconds between the transmission of line check messages. This definition applies only to Sender and Server Channels. Line check messages are used to determine if the remote partner is still connected. Line check messages will be sent while the connection is active, but only during the Disconnect Timeout interval.

The Line Check Timeout must be smaller than Disconnect Timeout. If it is set too small (that is, in the order of 1 - 5 seconds), the channel will be very busy sending and receiving line checks, and degrading MCA performance for other active channels. A number in the range of 15-30 seconds would be reasonable.

Another guideline is to make the Line Check Timeout value one-fourth the value of the Disconnect Timeout.

The default value for Line Check Timeout when creating a channel definition is “0”. With this value, line checks will not occur during the Disconnect Timeout interval. However, the Sender or Server MCA will still be able to detect and log the abnormal termination of a TCP/IP socket connection or an SNA connection.

Message Size

The Message Size field represents the maximum length of a message processed by this channel.

This field needs to be large enough to accommodate the largest message.

Therefore, if the anticipated largest message will be 2KB, the Max Message field should be 2KB (where KB equals 1024 bytes). If the largest message size is unknown, the default value of almost 64KB should be sufficient, but may be inefficient. As each installation is site-dependent, the best value will vary.

The default value for Max Message when creating a channel definition is the value specified for this field in the Queue Manager configuration.

MSN Wrap Count

The MSN Wrap Count field represents the highest MSN (Message Sequence Number) value which will be used on this channel, after which it will revert to 1.

The value of the MSN Wrap Count must be the same at both the sending and receiving ends of the channel.

The default value for the MSN Wrap Count is “999999”. This value is the recommended value.

Queue manager configuration guidelines

When configuring the Queue Manager, use the following guidelines for the following fields:

Max Message

The Max Message field represents the maximum length of a message processed by this MQSeries installation. No individual queue may have a Max Message size greater than that defined in the Queue Manager.

This field needs to be large enough to accommodate the largest message. This value should not be any larger than necessary, since any messages that are placed on the dead letter queue will also use up this amount of space on the disk (the dead letter queue is created automatically with this Max Message size as soon as the Queue Manager record is defined).

Therefore, if the anticipated largest message will be 2KB, then the Max Message field should be 2KB. If the largest message size is unknown, the default value of almost 64KB should be sufficient, but may be inefficient. As each installation is site-dependent, the best value will vary.

Max Poll Time

The Max Poll Time field represents the time in milliseconds between polls of a message queue, when there is an MQGET call using the wait option.

If the application will not be using the MQGET call with the wait option, then the value for poll time has no usage and can be set to zero.

If the application will be using the MQGET call with the wait option, the Max Poll Time value is very significant. If the value is a high rate, in the order of 50 milliseconds, a lot of disk access will degrade system performance. If the value is a slow rate, in the order of 250 to 500 milliseconds, then the application may not respond fast enough to an arriving message. In addition, another application could potentially read the data during the wait interval, assuming the queue was opened for shared access.

The default value for the Max Poll Time when creating the Queue Manager definition is “100”, as this represents a good compromise, between system performance and slow application access.

CAUTION: Do not set this field value to “0”; severe performance degradation will occur.

Queue configuration guidelines

When configuring a queue, use the following guidelines for the following fields:

Max Queue Depth

The Max Queue Depth field represents the maximum number of messages allowed on this queue. A value of “999999999” means that the depth is unlimited.

The value should be set to the maximum number of messages expected to be queued before an application or MCA starts to read and process the queue. Adding an extra 25% as a safety factor should be sufficient.

If the value is set too low, the queue could fill up too fast. An MCA attempting to put a received message on a full queue will get back an error and will be forced to put the message on the dead letter queue or return a rejection to the remote MCA if the dead letter queue is full.

The default value for Max Queue Depth when creating a queue definition is “999999999”.

Max Message

The Max Message field represents the maximum length of a message processed on this queue.

This field needs to be large enough to accommodate the largest message.

Therefore, if the anticipated largest message size will be 2KB, the Max Message field should be 2KB. If the largest message size is unknown, the default value of almost 64KB should be sufficient, but may be inefficient. As each installation is site-dependent, the best value will vary.

The default value for Max Message when creating a queue definition is the value specified for this field in the Queue Manager configuration.

Auto Purge

The Auto Purge field indicates whether or not automatic purging of deleted messages will occur on a queue. When the value “Y” or “L” is selected, a purge occurs when new messages are written to a queue under the following conditions:

- The queue depth is zero, AND
- Deleted messages exist in this queue, AND
- The number of current queue users is limited to the current process, which is putting a new message on the queue (The Monitor Local Queues screen will show a USERS count of 0).

The value “Y”, means that the system will always purge queues when possible. The value “L”, means that the system will only purge queues during the times specified by the “From” and “To” fields.

It is desirable to have Auto Purge enabled so that disk space is freed up automatically without operator intervention (otherwise this can be done using the MQM Purge Deleted Messages operations command). However, if there is a need to browse through deleted messages for a given queue, then Auto Purge should be disabled.

The default value for Auto Purge is “N” when creating a queue definition.

Number of channels per MCA guidelines

The MQSeries System has a limit of 9 MCA processes that can be supported, and a limit of 20 channels per MCA process. If it is necessary to run more than this number of channels, at least one MCA process must be configured to handle more than one channel.

One disadvantage to having multiple channels per MCA process is that if the MCA process is terminated (“killed” in UNIX terminology), then all the associated channels for that MCA process are also terminated.

Multiple MCA guidelines

As described earlier, if the channel count is greater than the maximum per MCA, it is necessary to use multiple MCAs.

Channel servicing is “multi-threaded”. Therefore, there is no difference in the way MQSeries System services channels regardless of whether the channels are defined to one or more MCAs.

Kernel configuration guidelines

It may be necessary to tune the kernel interprocess communication parameter SHMSEG to obtain the 9 MCA process configuration limit (see “Configuration capacities” on page 44). If the SHMSEG value in /etc/conf/cf.d/stune is less than 9, use the utility `idtune` to set it to at least that value (see the UnixWare manual “*System Administration, System Performance Administration*”). The system must be rebooted if the kernel parameter is changed.

Example configuration:

In this example, the user will be using both SNA and TCP/IP Channels and has computed the maximum message size to be 8KB. The SNA channels will be connecting to a mainframe.

Reasonable MQSeries System configuration values would be:

For the SNA Channels:

Field	Value	Units
Disconnect Timeout	5	seconds
Retry Count	10000	integer
Reconnect Timer	30	seconds
Checkpoint Count	10	integer
Line Check Timeout	30	seconds

For the TCP/IP Channels:

Field	Value	Units
Disconnect Timeout	60	seconds
Retry Count	10000	integer
Reconnect Timer	30	seconds
Checkpoint Count	10	integer
Line Check Timeout	30	seconds

If too many disconnects occur, increase the Disconnect Timeout value.

For the Queue Manager:

Field	Value	Units
Max Message	8	KB
Max Poll Time	100	milliseconds

For the Queue :

Field	Value	Units
Max Queue Depth	999999999	integer
Max Message	8	KB
Auto Purge	Y	character

Chapter 5. System operation

This chapter will describe the system operation and administration functions available in IBM MQSeries for UnixWare. Most such functions are provided through the menu-driven, screen-oriented program called Message Queue Management (MQM).

The menus and display screens of MQM are organized in a hierarchy as depicted in the following diagram.

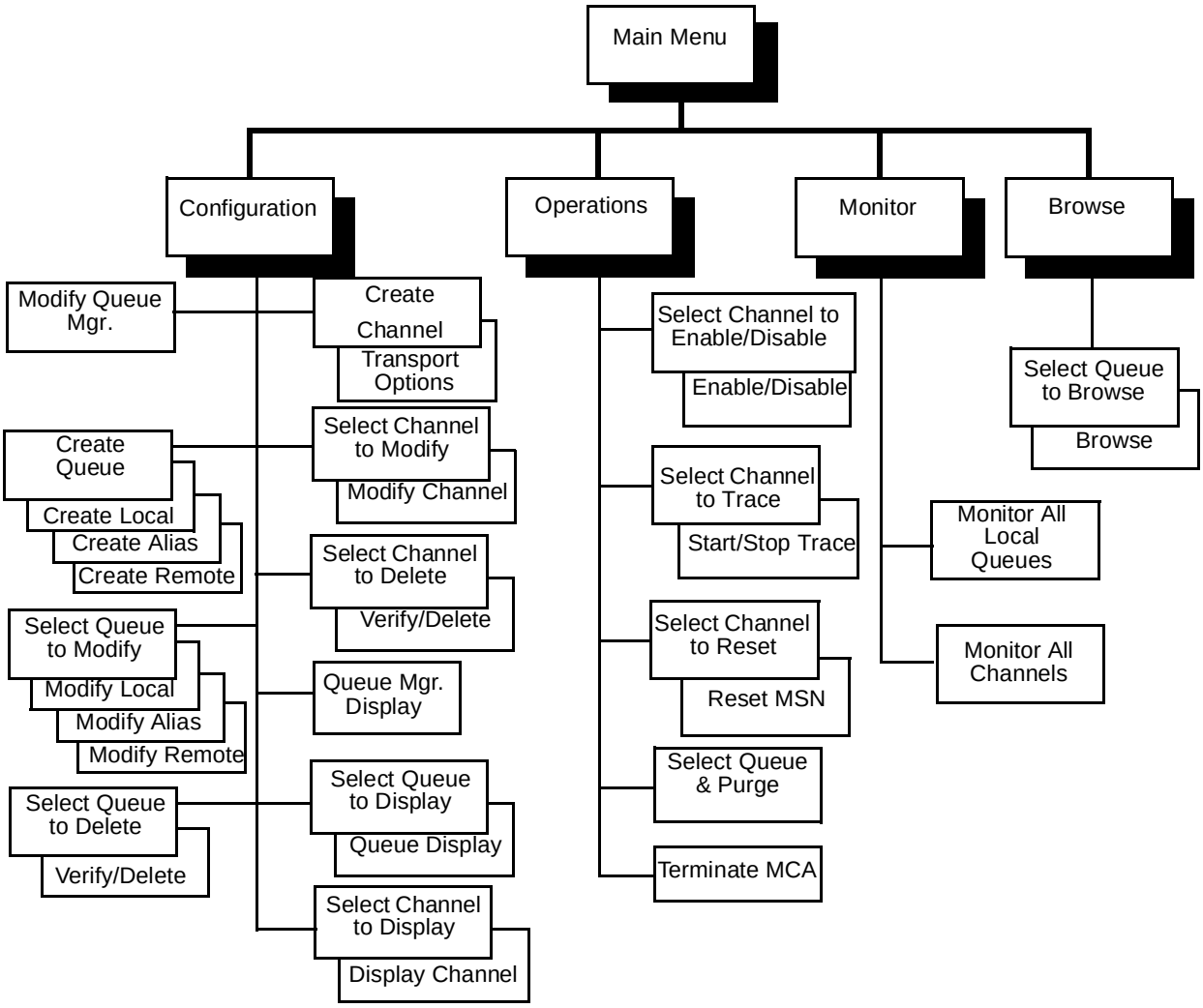


Figure 19. System administration relationships

In the next section, the main MQM menu is presented. The subsequent sections will present each of the operator functions available through these screens. The final section in this chapter will present those few functions which require operator action outside the MQM program.

IMPORTANT: After you have configured the system, or after having completed any significant portion of that effort, you should make a backup copy of the critical files, as follows:

```
cd <config.path>
cp SYSTEM.cdb  SYSTEM.cdb.SAV
cp SYSTEM.idx SYSTEM.idx.SAV
cp SYSTEM.qdt  SYSTEM.qdt.SAV
```

By performing this backup, you will have a “clean” copy of the channel database in case the original becomes corrupted, for any reason.

MQM operator interface - main menu

The MQSeries System administrator program, MQM, may be executed on any system at which the MQSeries System has been installed except for a "MQI application only" system (see "The MQSeries System's distributed architecture on UNIX" on page 3). To invoke MQM, simply type the following at the system prompt:

```
mqm
```

When MQM starts, the Main Menu is displayed.

```
                              IBM MQSeries for UnixWare Version 1
                                       ** Main Menu **


                                   Enter Choice:       1

                                   1.   Configuration

                                   2.   Operation

                                   3.   Monitoring

                                   4.   Browse QUEUE records



<return> - Select Option     <esc> - Discard Field Changes            CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Exit mqm
63H9503,5697-265 (C) Copyright IBM Corp. 1993, 1997 All Rights Reserved
```

Figure 20. Main menu

From the Main Menu, one of several sub-menus may be selected. The first three choices correspond to broad categories which include most MQSeries System operator functions:

- Configuring the MQSeries System
- Operating (controlling) the MQSeries System
- Monitoring the MQSeries System

The fourth function allows the operator to display the records on a selected queue.

- Browsing MQSeries System Queues

Each sub-menu presents a list of operator functions available from that screen. When a specific function is selected, the appropriate data entry or data display screens are presented to the operator.

Press <**Ctrl-X**> to exit MQM.

Note: There is one case in which MQM does not start at the Main Menu. The first time MQM is executed, it detects that no Message Queue Manager has been configured and branches directly to the Message Queue Manager definition screen. This might also happen if MQM cannot find the MQSeries System configuration files from the host workstation or if the MQSeries System configuration files have been corrupted.

Operator action keys

The action keys available on each MQSeries System operator screen are displayed at the bottom of the screen with an explanation of their function. These keys have been selected so that they are available from all types of keyboards, including ASCII terminals, and so that they conform to conventional UNIX key usage. For example, the cursor movement keys correspond to those used by the vi editor.

Configuration functions

Selecting option 1 (Configuration) from the Main Menu causes MQM to display the following sub-menu screen:

```
                          IBM MQSeries for UnixWare Version 1
                               ** Configuration Menu **

                              Enter Choice:       1

                              1.   Modify Queue Manager
                              2.   Display Queue Manager

                              3.   Create Queue
                              4.   Modify Queue
                              5.   Delete Queue
                              6.   Display Queue

                              7.   Create Channel
                              8.   Modify Channel
                              9.   Delete Channel
                              10.  Display Channel


<return> - Select Option     <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Go to previous menu
```

Figure 21. Configuration menu

From this screen, the operator can add, modify, delete, or display various MQSeries System configuration objects.

Press <**Ctrl-X**> to return to the Main Menu.

Modify queue manager

For each installation of the MQSeries System, one and only one Queue Manager must be defined. This is accomplished through the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                                  ** Queue Manager **


                Name: SYS1QMGR
         Description: Queue Manager for SYS1
         Config Path: /home/ssi/transact/comb/mqi
       Dead Letter Q: DEADLETTER
           Char. Set: 437
         Max Handles: 0
         Max Message: 1024
       Max Poll time: 100
        MCA Hostname:




<return> - FIELD EXIT        <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 22. Queue manager

On this screen, the data entry fields are as follows:

Name: This is the name of the local queue manager for this MQSeries System installation. The name may be up to 48 characters and must conform to the MQI naming requirements.

Description: This is a text field for operator use only. It may be up to 64 characters.

Config Path: This is the path to the disk directory which contains the MQSeries System configuration database.

Dead Letter Q: This is the name of the queue to which messages will be queued by the MQSeries System if they cannot be properly enqueued to their intended destination queue. Any messages directed to this queue will require operator action to recover.

Char. Set: This is a numeric field which identifies the Coded Character Set ID (CCSID) used by the Sun Solaris system. This value is used by the MQSeries System to determine the translation required for message headers between various hardware platforms on the network. The user data portion of messages is not translated.

This value must be any valid CCSID defined by IBM. Some of the CCSIDs recognized by MQSeries Systems follow:

Canada/French ASCII	863
Denmark/Norway EBCDIC	277
Finland/Sweden EBCDIC	278
France EBCDIC	297
Germany EBCDIC	273
Iceland EBCDIC	871
International EBCDIC	500
Italy EBCDIC	280
Latin Alphabet ASCII	819
Multilingual ASCII	850
Nordic ASCII	865
Portugal ASCII	860
Spain EBCDIC	284
UK EBCDIC	285
USA ASCII	437
USA EBCDIC	37

Note: *In the initial release of IBM MQSeries for UnixWare, characters used in any message header field must be restricted to the characters which are valid for queue names, as described in "Queue name format" on page 30.*

Max Handles: (This field is currently ignored.)

Max Message: This is a numeric field specifying the maximum length of a message processed by this MQSeries System installation. This length is used in the initial negotiation of communications channels to other MQSeries Systems. This will only affect queues created subsequent to the modification of this field.

Max Poll time: This is the time in msec between polls of a message queue, when there is an `MQGET` call with wait. The lower this number, the more frequently the disk drive is kept busy. Queue polling is used in the support of application triggering. (See "Triggering" on page 110.)

MCA Host name: This is the TCP/IP hostname of the system acting as the MQI server for the transmission of messages to remote systems. (See “The MCAMD process” on page 102.) If the queue manager being configured is on the MQI server system, the MCA Hostname field may be left blank. If the field is blank, the MQM utility will communicate with the MCAMD via UNIX domain sockets. If the field is not blank, then it should contain the TCP/IP hostname of the MQI server system (which must match a hostname given in the /etc/hosts file). The MQM utility will then communicate with the MCAMD via Internet domain sockets. If the MQM utility is executed on the MQI server system, it is more efficient to leave the field blank because there is less overhead in UNIX domain sockets than Internet domain sockets. If the MQM utility is executed on an “MQI application only” system, the field MUST contain the Internet hostname of the MQI server system.

Note: All MCAs must be terminated and restarted for the Queue Manager modification to take effect.

The screen may be exited with either <**Ctrl-X**>, to discard changes, or with <**Ctrl-W**>, to save changes. Both of these cause the return to the Configuration Menu.

Display queue manager

Choice 2 on the Configuration Menu allows an operator to view the attributes defined for the local queue manager through the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                              ** Display Queue Manager **


                 Name: SYS1QMGR
          Description: Queue Manager for SYS1
          Config Path: /home/ssi/transact/comb/mqi
        Dead Letter Q: DEADLETTER
            Char. Set: 437
          Max Handles: 0
          Max Message: 1024
        Max Poll time: 100
         MCA Hostname:



                           *** Press any key to continue ***
```

Figure 23. Display queue manager

This is a display only screen.

The operator may press any key to return to the Configuration Menu.

Create queue

Choice 3 on the Configuration Menu allows an operator to create queue definitions as required in order to configure the local installation of the MQSeries System.

To define a queue, two screens are involved. The first screen is the same for all queues. It allows entry of the queue *name* and *type*. Based on the *type* entered, the appropriate second screen is displayed for the operator to enter the remainder of the data to complete the definition. The first screen displayed is:

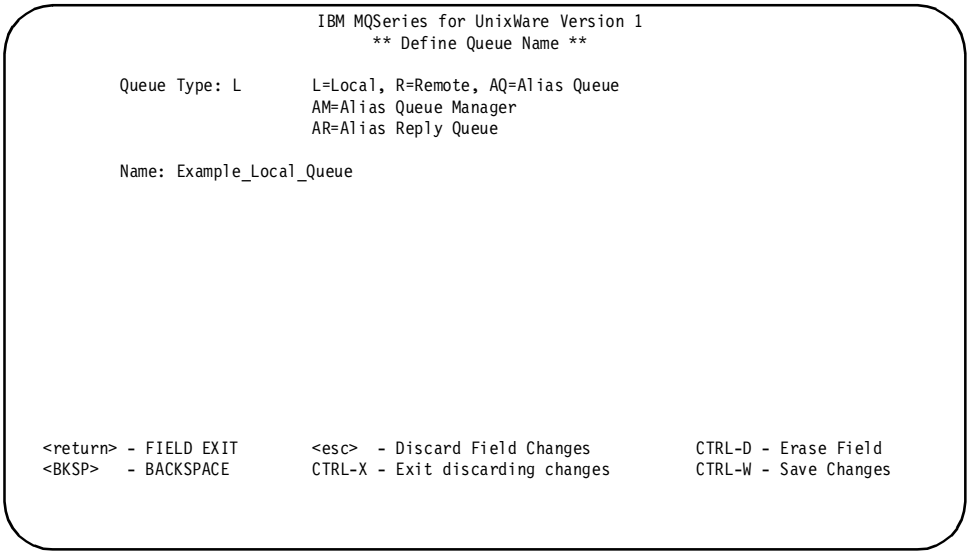


Figure 24. Define queues

On this screen, the data entry fields are:

Queue Type: This is a two-character field with the acceptable entries listed on the screen. The type determines the next screen to be displayed.

Name: This is the name of the queue (or alias) being defined. The name may be up to 48 characters, must be unique among all other defined queues for this installation, and must conform to the MQI naming requirements.

Upon entry of the above two fields, one of the following screens is displayed:

Create local queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Create Local Queue **



                    Name: Example_Local_Queue
             Description: This is an example local queue
                   Usage: 0    0 = Normal, 1 = Transmission
               File Name: exlocque

           Max Queue Depth: 100
               Max Message: 1024

                 Auto Purge: L       Y - Yes         N - No         L - Limit
                        From: 14:31            To: 04:59



<return> - FIELD EXIT        <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 25. Create local queue

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Text field for operator use only. It may be up to 64 characters.

Usage: *Normal* means the queue is used by an application to receive inbound messages. *Transmission* means the queue is used by the MQSeries System to hold outbound messages destined for another MQSeries System queue manager.

File Name: The 8-character field indicating the UNIX file name used to store messages for this queue. This must be a valid filename and ".que" is appended.

Max Queue Depth: The maximum number of messages allowed on this queue. A value of 999999999 means depth is unlimited.

Max Message: The maximum length of application data in a message processed on this queue.

Note: The value chosen for the Max Message size can affect performance (messages per second and/or CPU time per message). For example, if the Max Message size is set to 64,000 bytes, but only 100 byte messages are sent to a queue, it can cost the same in elapsed and CPU time to `MQPUT` or `MQGET` to the queue as it would have if every message were a full 64,000 bytes.

Auto Purge: This field can only be filled in with either a 'Y', 'N', or 'L'. These values mean that:

Y - Yes, always purge queues when possible.
N - No, do not automatically purge queues.
L - Yes, but only during the times specified by the From and To fields.

When Y or L is selected, purges occur when new messages are written to a queue under the following conditions:

- the queue depth is 0, and
- deleted messages exist in this queue, and
- the number of current users is limited to the current process only.

Note that for an active queue, these conditions could mean that messages are not auto purged. The presence of unpurged messages on a queue can affect performance.

From: This field (though always editable) is only used when Auto Purge is set to 'L'. It must be specified in 24-hour clock format (HH:MM). This is the beginning time for automatic queue purging.

To: This field (though always editable) is only used when Auto Purge is set to 'L'. It must be specified in 24-hour clock format (HH:MM). This is the ending time for automatic queue purging. This value may be less than the From time if automatic queue purging is to begin prior to midnight and continue through morning.

Note: Two fields, Inhibit Get and Inhibit Put, are not displayed but are set to default values of 'N'. They can be modified on the Modify Local Queue screen. (See "Modify local queue" on page 70.)

Upon exiting this screen, the program returns to the Configuration Menu.

Create remote queue

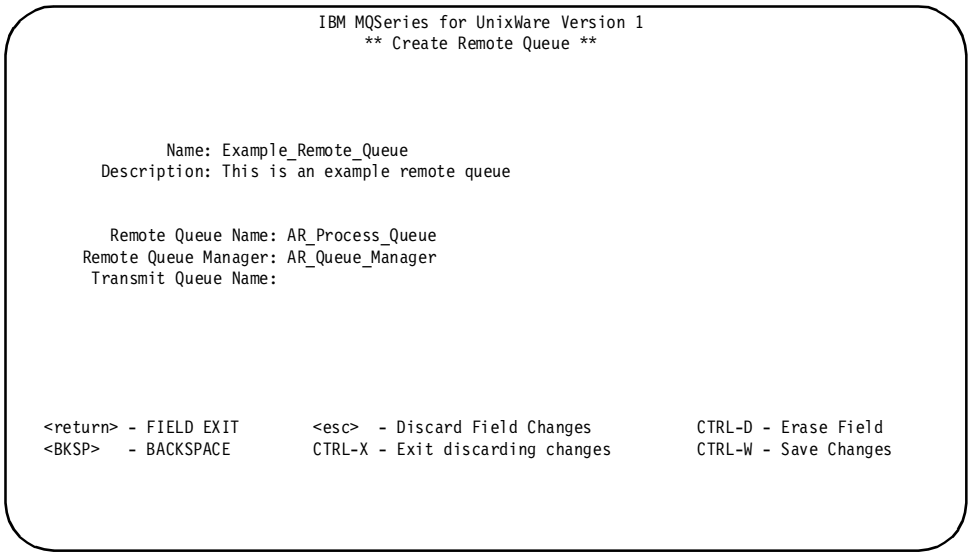


Figure 26. Create remote queue

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Text field for operator use only. It may be up to 64 characters.

Remote Queue Name: The queue name on the remote MQSeries System to which the definition in progress will refer.

Remote Queue Manager: The name of the remote MQSeries System on which *Remote Queue Name* is defined as a local queue. This name must be defined as a local transmission queue unless the following field is used.

Transmit Queue Name: The name of the local transmission queue to be used by MQSeries System to convey messages to this remote queue. If left blank then the *Remote Queue Manager* is required to map to a local transmission queue.

Upon exiting this screen, the program returns to the Configuration Menu.

Create alias queue

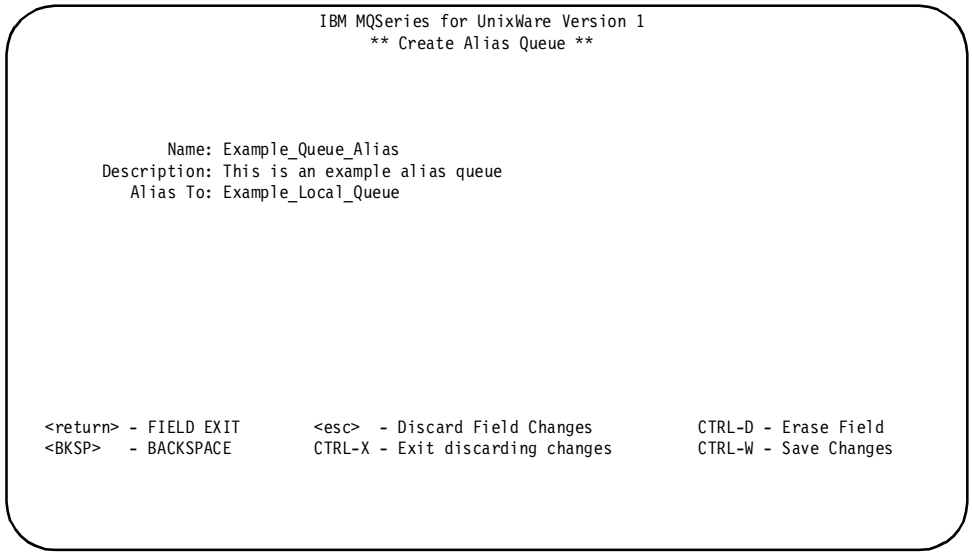


Figure 27. Create alias queue

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Text field for operator use only. It may be up to 64 characters.

Alias To: The name of another object already defined in the local configuration. This can be a *local queue name*, or a *remote queue name*. It cannot identify another *alias*.

Upon exiting this screen, the program returns to the Configuration Menu.

Create alias queue manager

```
                          IBM MQSeries for UnixWare Version 1
                            ** Create Alias Queue Manager **



                Name: Example_Alias
         Description: This is an example alias queue manager
            Alias To: Example_Queue_Manager

 Transmit Queue Name:









 <return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
 <BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 28. Create alias queue manager

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Text field for operator use only. It may be up to 64 characters.

Alias To: The name of a known queue manager. This can be a *local transmit queue name*, a *remote queue manager name*, or the *local queue manager name*. It cannot identify another *alias*.

This name must be the local queue manager or a local transmission queue unless the following field is used.

Transmit Queue Name: The name of the local transmission queue to be used by the MQSeries System to convey messages to this remote queue manager. If left blank, then the **Alias To:** field is required to map to a local transmission queue or to the local queue manager name.

Upon exiting this screen, the program returns to the Configuration Menu.

Create alias reply queue

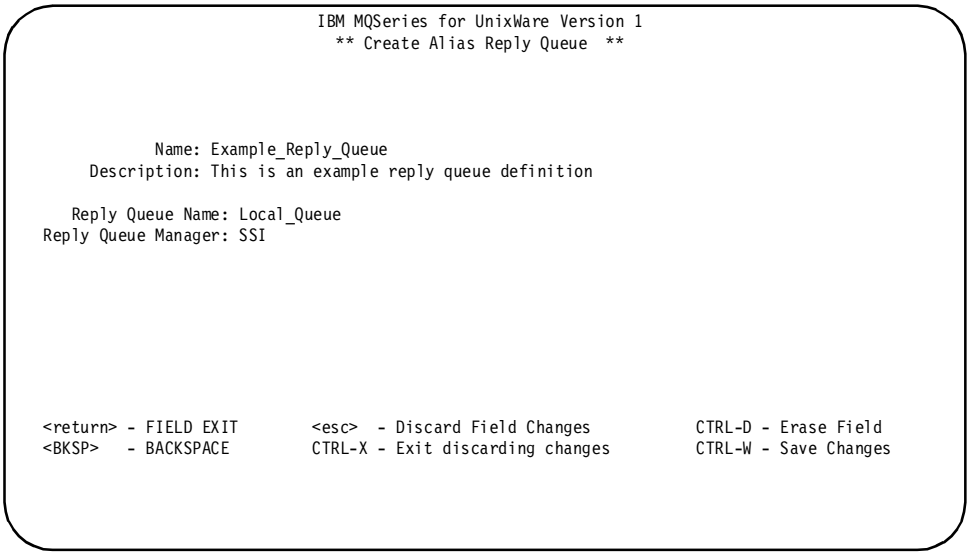


Figure 29. Create alias reply queue

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Text field for operator use only. It may be up to 64 characters.

Reply Queue Name: This is the queue name that will replace the ReplyToQ field in the MQMD.

Reply Queue Manager: This is the queue manager name that will replace the ReplyToQMgr field in the MQMD.

Upon exiting this screen, the program returns to the Configuration Menu.

Modify queue

Choice 4 on the Configuration Menu allows an operator to modify existing queue definitions.

To modify a queue, two screens are involved. The first allows the operator to select the desired queue from a list of defined queues. Based on the selection, the appropriate second screen is displayed for the operator to enter the remainder of the data to complete the modification. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                              ** Select Queue to Modify**

Local_Queue                                LOCAL
Queue_Alias                                ALIAS
To_VMS_Queue                               REMOTE
Transmit_Queue                             TRANSMIT
DEADLETTER                                 LOCAL
DefMan                                     MGR ALIAS
Example_Local_Queue                        LOCAL
Example_Queue_Alias                        ALIAS
Example_Remote_Queue                       REMOTE
VAXINP                                     TRANSMIT
VAXOUT                                     LOCAL
VAXQMGR                                    TRANSMIT
VMS_Queue_Manager                          TRANSMIT



J       - Down              K         -   Up              <return>   - Select
CTRL-F  - PgDn              CTRL-B    - PgUp              CTRL-X     - Exit
```

Figure 30. Select queue to modify

On this screen, each line displays the name and type of a defined queue. The operator uses the cursor control keys to highlight the desired queue and then presses <**Return**> to select the queue. The operator may also press <**Ctrl-X**> to return to the Configuration Menu. Based on the attributes of the selected queue, one of the following screens is displayed:

Modify local queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Modify Local Queue **



              Name: Example_Local_Queue
       Description: This is an example local queue
             Usage: 0   0 = Normal, 1 = Transmission
         File Name: exlocq
       Max Message: 1024


 Max Queue Depth: 0
Inhibit Get[Y/N]: N
Inhibit Put[Y/N]: N
      Auto Purge: L          Y - Yes        N - No         L - Limit
             From: 14:31              To: 04:59



<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 31. Modify local queue

On this screen, the data entry fields are:

Name:	Filled in from the previous screen. Cannot be modified.
Description:	Same as during queue creation. Cannot be modified.
Usage:	Same as during queue creation. Cannot be modified.
File Name:	Same as during queue creation. Cannot be modified.
Max Message:	Same as during queue creation. Cannot be modified.
Max Queue Depth:	Same as during queue creation.
Inhibit Get:	This is a toggle which enables/disables `MQGET` operations against this queue. `MQGET` retrieves a message from a local queue that has been opened with an `MQOPEN` call.
Inhibit Put:	This is a toggle which enables/disables `MQPUT` operations against this queue. `MQPUT` puts a message on a queue that has been opened with an `MQOPEN` call
Auto Purge:	Same as during queue creation. Processes that have the queue open will recognize this modification.
From:	Same as during queue creation.
To:	Same as during queue creation.

Upon exiting this screen, the program returns to the Select Queue to Modify screen.

Modify remote queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Modify Remote Queue **



                 Name: Example_Remote_Queue
            Description: This is an example remote queue


              Remote Queue Name: AR_Process_Queue
           Remote Queue Manager: AR_Queue_Manager
            Transmit Queue Name:
               Inhibit Put[Y/N]: N




<return> - FIELD EXIT       <esc>  - Discard Field Changes        CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes      CTRL-W - Save Changes
```

Figure 32. Modify remote queue

On this screen, the data entry fields are:

Name:	Filled in from the previous screen. Cannot be modified.
Description:	Same as during queue creation. Cannot be modified.
Remote Queue Name:	Same as during queue creation. Cannot be modified.
Remote Queue Manager:	Same as during queue creation. Cannot be modified.
Transmit Queue Name:	Same as during queue creation. Cannot be modified.
Inhibit Put:	Same as for a local queue.

Upon exiting this screen, the program returns to the Select Queue to Modify screen.

Modify alias queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Modify Alias Queue **



                Name: Example_Queue_Alias
         Description: This is an example alias queue
            Alias To: Example_Local_Queue


          Inhibit Get[Y/N]: N
          Inhibit Put[Y/N]: N




<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 33. Modify alias queue

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Same as during queue creation. Cannot be modified.

Alias To: Same as during queue creation. Cannot be modified.

Inhibit Get: Same as for a local queue.

Inhibit Put: Same as for a local queue.

Upon exiting this screen, the program returns to the Select Queue to Modify screen.

Modify alias queue manager

```
                          IBM MQSeries for UnixWare Version 1
                            ** Modify Alias Queue Manager**



                Name: Example_Alias
         Description: This is an example alias queue manager
            Alias To: Example_Queue_Manager

Transmit Queue Name:









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 34. Modify alias queue manager

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Same as during queue manager creation.

Alias To: Same as during queue manager creation. Cannot be modified.

Transmit Queue Name: Same as during queue manager creation. Cannot be modified.

Upon exiting this screen, the program returns to the Select Queue to Modify screen.

Modify alias reply queue

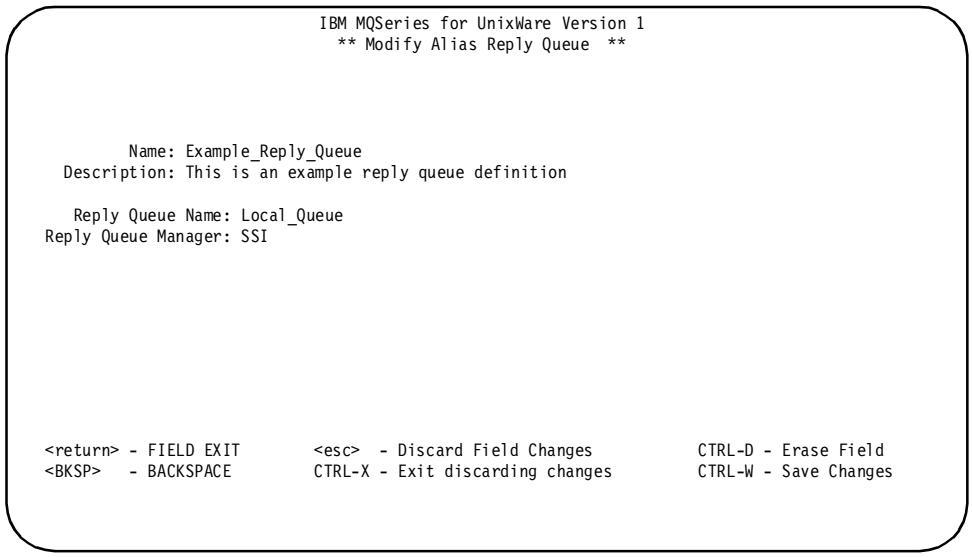


Figure 35. Modify alias reply queue

On this screen, the data entry fields are:

Name: Filled in from the previous screen. Cannot be modified.

Description: Same as during queue manager creation. Cannot be modified.

Reply Queue Name: This is the queue name that will replace the ReplyToQ field in the MQMD. Cannot be modified.

Reply Queue Manager: This is the queue manager name that will replace the ReplyToQMgr field in the MQMD. Cannot be modified.

Upon exiting this screen, the program returns to the Select Queue to Modify screen.

Delete queue

Choice 5 on the Configuration Menu allows an operator to delete existing queue definitions.

To delete a queue, two screens are involved. The first allows the operator to select the desired queue from a list of defined queues. Based on the selection, the selected queue definition is displayed on an appropriate second screen. On this screen, the operator is asked to verify the delete request. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                             ** Select Queue to Delete **

Local_Queue                                LOCAL
Queue_Alias                                ALIAS
To_VMS_Queue                               REMOTE
Transmit_Queue                             TRANSMIT
DEADLETTER                                 LOCAL
DefMan                                     MGR ALIAS
Example_Local_Queue                        LOCAL
Example_Queue_Alias                        ALIAS
Example_Remote_Queue                       REMOTE
UNIQUE_Local_Queue                         LOCAL
VAXINP                                     TRANSMIT
VAXOUT                                     LOCAL
VAXQMGR                                    TRANSMIT
VMS_Queue_Manager                          TRANSMIT



J       - Down              K         -   Up              <return>   - Select
CTRL-F  - PgDn              CTRL-B    - PgUp              CTRL-X     - Exit
```

Figure 36. Delete queue

On this screen, the operator uses the cursor control keys to highlight the desired queue and then presses <**Return**> to select the queue. The operator may also press <**Ctrl-X**> to return to the Configuration Menu. Based on the attributes of the selected queue, one of the following screens is displayed:

Delete local queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Delete Local Queue **


            Name: Local_Queue
     Description: This is an example local queue
           Usage: 0   0 = Normal,   1 = Transmission
       File Name: locq1
     Max Message: 1024


 Max Queue Depth: 100
Inhibit Get[Y/N]: N
Inhibit Put[Y/N]: N
      Auto Purge: N          Y - Yes         N - No         L - Limit
            From: 14:31              To: 04:59

        Is this the queue you wish to delete? [Y/N]:N


<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 37. Delete local queue

This screen displays the parameters of the local queue definition which has been selected for deletion and prompts the operator to verify the delete request.

The operator responds:

Y to delete the displayed queue

N to abort the delete request

Upon exiting this screen, the program returns to the Select Queue to Delete screen.

Delete remote queue

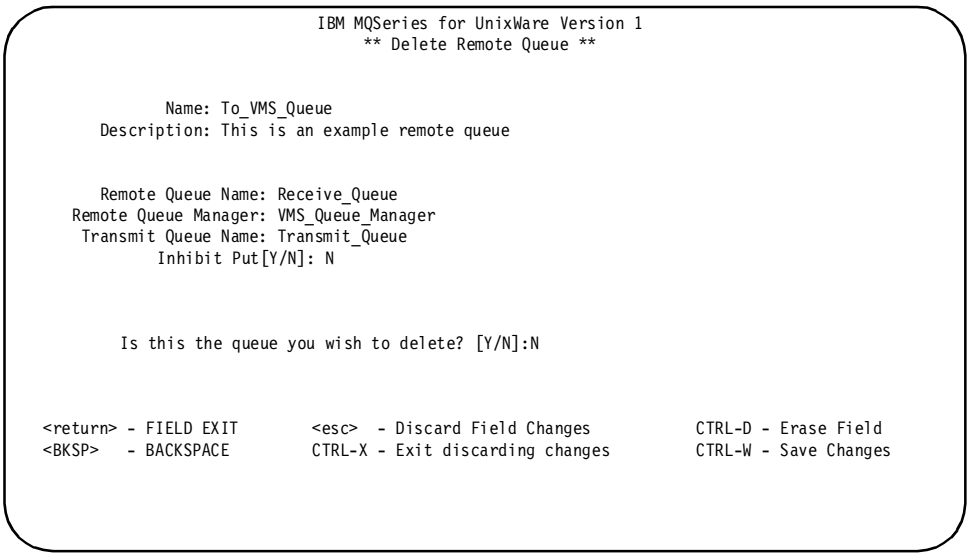


Figure 38. Delete remote queue

This screen displays the parameters of the remote queue definition which has been selected for deletion and prompts the operator to verify the delete request.

The operator responds:

Y to delete the displayed queue

N to abort the delete request

Upon exiting this screen, the program returns to the Select Queue to Delete screen.

Delete alias queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Delete Alias Queue **


                Name: Queue_Alias
         Description: This is an example queue alias
            Alias To: Local_Queue

Inhibit Get[Y/N]: N
Inhibit Put[Y/N]: N


          Is this the queue you wish to delete? [Y/N]:N




<return> - FIELD EXIT        <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 39. Delete alias queue

This screen displays the parameters of the alias definition which has been selected for deletion and prompts the operator to verify the delete request.

The operator responds:

Y to delete the displayed queue

N to abort the delete request

Upon exiting this screen, the program returns to the Select Queue to Delete screen.

Delete alias queue manager

```
                          IBM MQSeries for UnixWare Version 1
                            ** Delete Alias Queue Manager **


                Name: Manager_Alias
         Description: This is an example queue manager alias
            Alias To: Local_Queue_Manager

    Transmit Queue Name:



          Is this the alias you wish to delete? [Y/N]:N




<return> - FIELD EXIT        <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 40. Delete alias queue manager

This screen displays the parameters of the alias definition which has been selected for deletion and prompts the operator to verify the delete request.

The operator responds:

Y to delete the displayed queue manager alias

N to abort the delete request

Upon exiting this screen, the program returns to the Select Queue to Delete screen.

Delete alias reply queue

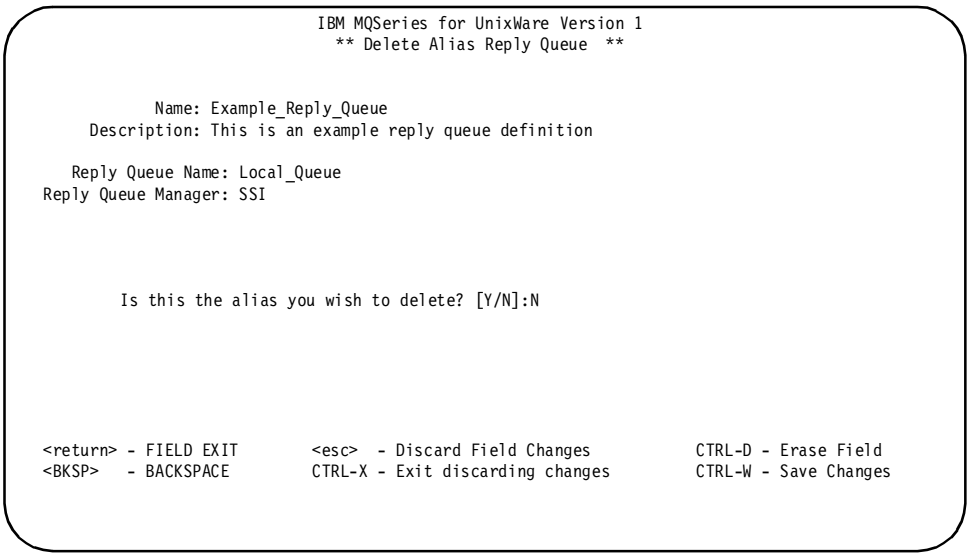


Figure 41. Delete alias reply queue

This screen displays the parameters of the alias definition which has been selected for deletion and prompts the operator to verify the delete request.

The operator responds:

Y to delete the displayed queue manager alias

N to abort the delete request

Upon exiting this screen, the program returns to the Select Queue to Delete screen.

Display queue

Choice 6 on the Configuration Menu allows an operator to view existing queue definitions.

Note: This function allows an operator to see the queue definition, not the current queue status. To see the current queue status, refer to the Monitor Queues function in "Monitor queues" on page 97.

To view a queue definition, two screens are involved. The first allows the operator to select the desired queue from a list of defined queues. Based on the selection, the selected queue definition is displayed on an appropriate second screen. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                             ** Select Queue to Display **

Local_Queue                               LOCAL
Queue_Alias                               ALIAS
To_VMS_Queue                              REMOTE
Transmit_Queue                            TRANSMIT
DEADLETTER                                LOCAL
DefMan                                    MGR ALIAS
Example_Local_Queue                       LOCAL
Example_Queue_Alias                       ALIAS
Example_Remote_Queue                      REMOTE
UNIQUE_Local_Queue                        LOCAL
VAXINP                                    TRANSMIT
VAXOUT                                    LOCAL
VAXQMGR                                   TRANSMIT
VMS_Queue_Manager                         TRANSMIT



J       - Down              K         -   Up              <return>   - Select
CTRL-F  - PgDn              CTRL-B    - PgUp              CTRL-X     - Exit
```

Figure 42. Select queue to display

On this screen, each line displays the name and type of a defined queue. The operator uses the cursor control keys to highlight the desired queue and then presses **<Return>** to select the queue. The operator may also press **<Ctrl-X>** to return to the Configuration Menu. Based on the attributes of the selected queue, one of the following screens is displayed:

Display local queue

```
                          IBM MQSeries for UnixWare Version 1
                               ** Display Local Queue **



             Name: Local_Queue
      Description: This is an example local queue
            Usage: 0 0 = Normal, 1 = Transmission
        File Name: locq1
      Max Message: 1024


 Max Queue Depth: 100
Inhibit Get[Y/N]: N
Inhibit Put[Y/N]: N

      Auto Purge: L          Y - Yes         N - No         L - Limit
              From: 14:31              To: 04:59

                  *** Press any key to continue ***
```

Figure 43. Display local queue

This is a display only screen.

The operator may press any key to return to the Select Queue to Display screen.

Display remote queue

```
                              IBM MQSeries for UnixWare Version 1
                                   ** Display Remote Queue **



                     Name: To_VMS_Queue
                Description: This is an example remote queue


                Remote Queue Name: Receive_Queue
              Remote Queue Manager: VMS_Queue_Manager
               Transmit Queue Name: Transmit_Queue
                      Inhibit Put[Y/N]: N




                           *** Press any key to continue ***
```

Figure 44. Display remote queue

This is a display only screen.

The operator may press any key to return to the Select Queue to Display screen.

Display alias queue

```
                              IBM MQSeries for UnixWare Version 1
                                   ** Display Alias Queue **


                     Name: Queue_Alias
                Description: This is an example queue alias
                   Alias To: Local_Queue

          Inhibit Get[Y/N]: N
          Inhibit Put[Y/N]: N




                        *** Press any key to continue ***
```

Figure 45. Display alias queue

This is a display only screen.

The operator may press any key to return to the Select Queue to Display screen.

Display alias queue manager

```
                          IBM MQSeries for UnixWare Version 1
                           ** Display Alias Queue Manager **


                Name: Manager_Alias
         Description: This is an example queue manager alias
            Alias To: Local_Queue_Manager

     Transmit Queue Name:




                *** Press any key to continue ***
```

Figure 46. Display alias queue manager

This is a display only screen.

The operator may press any key to return to the Select Queue to Display screen.

Display alias reply queue

```
                          IBM MQSeries for UnixWare Version 1
                           ** Display Alias Reply Queue  **

           Name: Example_Reply_Queue
    Description: This is an example reply queue definition

    Reply Queue Name: Local_Queue
 Reply Queue Manager: SSI




                *** Press any key to continue ***
```

Figure 47. Display alias reply queue

This is a display only screen.

The operator may press any key to return to the Select Queue to Display screen.

Create channel

Choice 7 on the Configuration Menu allows an operator to create channel definitions as required in order to configure the local installation of the MQSeries System.

Note: While MQM can operate successfully on queue information without the MCAMD process, the MCAMD is required to access or modify channel information.

To define a channel, two screens are involved. The first is the same for all channels. It allows the operator to enter generic parameters such as, channel name, message size, etc. The second screen varies according to the transport protocol selected for the channel.

The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                                ** Create Channel **

       Channel Name: SYS1.TO.MVS1                MSN: 1
         Queue Name: Transmit_Queue
               Type: 1      1 = Sender 2 = Server 3 = Receiver 4 = Requester

* * * M A X I M U M S * * *                        * * * T I M E R S * * *
        Retry Count: 1000                         Disconnect: 300
     MSN Wrap Count: 999999                       Reconnect: 30
   Checkpoint Count: 10                           Line Check: 20
       Message Size: 2048

        Transport Protocol: 0          0 = LU6.2 1 = TCP/IP




<return> - FIELD EXIT        <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 48. Create channel

On this screen, the data entry fields are:

Channel Name: The name of the Channel to be created. A maximum of 20 characters is allowed. Acceptable characters are the same as for queue names. A channel must have the same name on each side of the connection. See “Queue name format” on page 30.

MSN: Initial value for the Message Sequence Number.

Queue Name: If the channel is a SENDER or SERVER, this specifies the name of the transmit queue. This field is not used by a RECEIVER or REQUESTER.

Type: The type of channel being created. A channel type is defined by the direction of data flow and whether the process is initiated locally or via the complementary process remotely. Using SNA terminology, these are typically referred to as the Source and Target Transaction Programs. Therefore, there are four possible combinations:

SENDER	Originates data and initiates remote RECEIVER.
RECEIVER	Receives data, initiated by SENDER.
REQUESTER	Receives data and initiates remote SERVER.
SERVER	Originates data, initiated by REQUESTER.

Retry Count: The number of times a connection is retried either when communications have not been established, or when communications have been established but have subsequently failed. Each retry is performed at the interval specified in the Reconnect field (see **Reconnect**).

MSN Wrap Count: The highest MSN value which will be used on this channel, after which it will revert to 1. The value coded must be in the range 1 to 999999. Code the same value on the definitions of this channel at both the sending and receiving ends of the channel; if the values differ, the MSN Wrap Count will NOT be negotiated during channel initialization: the channel will close without messages being transferred.

Checkpoint Count: The number of messages that will be sent before a checkpoint, or "snapshot", of the channel information will be taken to the system channel database.

Message Size: The maximum size of a message in bytes. The value must be less than or equal to the maximum message value specified for the local queue manager. (See "Modify queue manager" on page 61.) Other systems ask for maximum transmit size. For UNIX, the maximum transmit size is Message Size plus 476 bytes.

Disconnect: The length of time in seconds before a disconnect due to an idle (empty) transmit queue. This interval allows for the possibility of new messages getting enqueued and transmitted before the channel disconnects. A Sender which is idle will connect (or reconnect) when more messages arrive on the transmit queue. A Server which is idle will not connect (or reconnect) until the corresponding Requester starts (or restarts) communication. If it is necessary to prevent the channel from connecting (or reconnecting), the Disable Channel command should be used. A value of zero designates no disconnect. To disconnect such a channel, use the Disable Channel command. This definition applies only to Sender and Server Channels.

Reconnect: The length of time in seconds before a reconnect is attempted either when communications have not been established, or when communications have been established but have subsequently failed. A reconnect is attempted the number of times specified in the Retry Count field (see **Retry Count**). A value of 0 (zero) designates no reconnect.

Line Check: The length of time in seconds between the transmission of line check messages. This definition applies only to Sender and Server Channels. Line check messages are used to determine if the remote partner is still connected. They are sent during the disconnect time-out interval.

Transport Protocol: A number defining the type of transport. In this case, only a 0 (zero) or a 1 (one) is allowed designating SNA Type LU 6.2 or TCP/IP, respectively.

Upon exiting this screen with <**Ctrl-W**>, the second screen is displayed to allow entry of transport protocol parameters. For an LU 6.2 channel (for example, SENDER channel), this screen is:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Create Channel (LU 6.2 Parameters)**

    Channel Name: SYS1.TO.MVS1                    Type: 1 SENDER

     Symbolic Destination Name:
                     Mode Name:    DFHLU62
                Source LU Name:    UWLU
               Partner LU Name:    MVSLU
      Transaction Program Name:    MVSRCV









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 49. Create channel - LU 6.2 parameters (SENDER channel)

On this screen, the data entry fields are:

Symbolic Destination Name: This is the name (limited to 8 characters) of the index into the "Side Information" database. A <return> skips this field.

If the above parameter is skipped, the following four parameters have to be specified:

Source LU Name A name specifying the local LU (limited to 17 characters) that may either be fully-qualified or locally known.

Partner LU Name A name specifying the LU on the remote system (limited to 17 characters) that may either be fully-qualified or locally known.

Mode Name The name of the mode (limited to 8 characters) used by LU 6.2 to designate the properties for the session that will be allocated for the conversation. To use the BLANK mode, specify "null".

TP Name The name of the partner transaction program on the remote system (limited to 64 characters).

Upon exiting this screen, the program returns to the Configuration Menu.

Upon exiting the Create Channel screen with <**Ctrl-W**>, the second screen for a TCP/IP channel is:

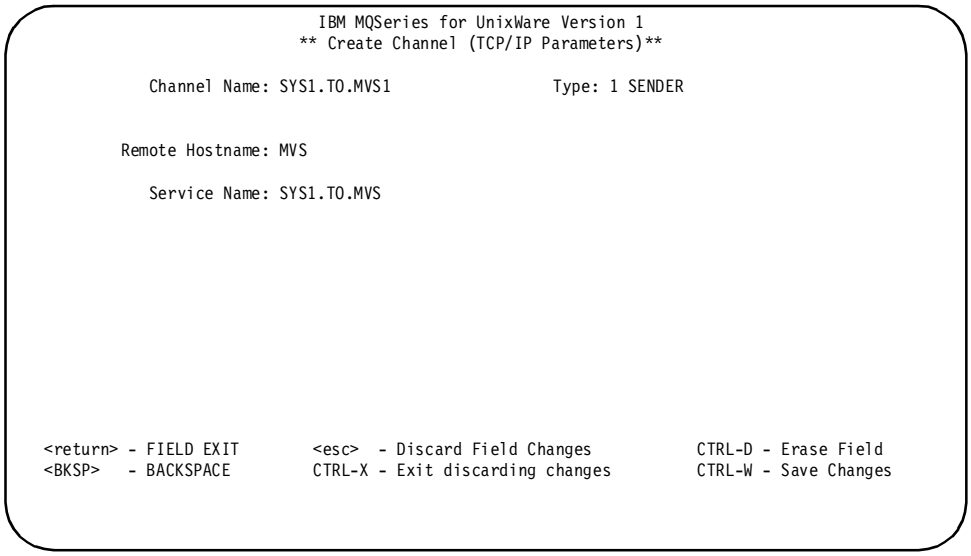


Figure 50. Create channel (TCP/IP parameters)

On this screen, the data entry fields are:

Remote Hostname: The name of host to which we need to connect, and only applies to SENDER and REQUESTER channels. (See “MQSeries System utilizing TCP/IP” on page 36.)

Note: The local MCA Host Name is defined on the Modify Queue Manager screen. (See “Modify queue manager” on page 61.)

Service Name: The name of TCP/IP service defined in the `/etc/services` file. For every channel defined using TCP/IP as a transport, you should define a Service Name in the file `/etc/services`. This entry must also exist on the remote system. (See “MQSeries System utilizing TCP/IP” on page 36.)

Upon exiting this screen, the program returns to the Configuration Menu.

Modify channel

Choice 8 on the Configuration Menu allows an operator to modify existing channel definitions.

A channel must be disabled before it can be modified. For modifications to Queue Name, Type, and Transport parameter to take effect and work properly, the MCA process must be stopped and restarted. To modify a channel, three screens are involved. The first allows the operator to select the desired channel from a list of defined channels. The parameters of the selected channel are then displayed for the operator to modify. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                            ** Select Channel to Modify **


Channel Name            TYPE            Status          PID       MSN             Trace
================================================================================================
SYS1.TO.MVS1            SENDER          DISABLED        0000000   0000416         OFF
MVS1.TO.SYS1            REQUESTER       DISABLED        0000000   0004077         OFF
VAXOUT                  RECEIVER        IDLE            0014209   0000011         OFF
VAXINP                  SERVER          DISABLED        0014209   0000031         OFF
TO.MVS1                 SERVER          DISABLED        0000000   0000001         OFF
MVS1.TO                 RECEIVER        DISABLED        0000000   0001715         OFF
SYS1.TO.VSE2            SENDER          DISABLED        0000000   0000041         ON
VSE2.TO.SYS1            REQUESTER       DISABLED        0000000   0000021         ON
TCPIP.TEXT.1            RECEIVER        DISABLED        0000000   0000257         ON




J       - Down                  K         -   Up                  <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 51. Select channel to modify

On this screen, each line displays the name and type of a defined channel. The operator uses the cursor control keys to highlight the desired channel and then presses <**Return**> to select the channel. Press <**Ctrl-X**> to return to the Configuration Menu. The selected channel is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                                ** Modify Channel **

       Channel Name: SYS1.TO.MVS1                MSN: 1
         Queue Name: Transmit_Queue
               Type: 1      1 = Sender 2 = Server 3 = Receiver 4 = Requester

* * * M A X I M U M S * * *                     * * * T I M E R S * * *
         Retry Count:  1000                     Disconnect: 300
       MSN Wrap Count:  999999                   Reconnect: 30
    Checkpoint Count:  10                       Line Check: 20
         Message Size:  2048



   Transport Protocol: 0              0 = LU6.2      1 = TCP/IP




<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 52. Modify channel

On this screen, the data entry fields are the same as those for creating a channel. See “Create channel” on page 81.

Upon exiting this screen with <**Ctrl-W**>, another screen is displayed to allow modifications of transport protocol parameters. For an LU 6.2 channel (for example, SENDER), this screen is:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Modify Channel (LU 6.2 Parameters)**

    Channel Name: SYS1.TO.MVS1                    Type: 1 SENDER

    Symbolic Destination Name:
                   Mode Name:     DFHLU62
             Source LU Name:      LWLU
            Partner LU Name:      MVSLU
      Transaction Program Name:   MVSRCV









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 53. Modify channel - LU 6.2 parameters (SENDER channel)

On this screen, the data entry fields are the same as those for creating a channel. See “Create channel” on page 81.

Upon exiting this screen, the program returns to the Select Channel to Modify screen.

Upon exiting the Modify Channel screen with <**Ctrl-W**>, the second screen for a TCP/IP channel is:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Modify Channel (TCP/IP Parameters)**

           Channel Name: SYS1.TO.MVS1                Type: 1 SENDER


         Remote Hostname: MVS

           Service Name: SYS1.TO.MVS









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 54. Modify channel (TCP/IP parameters)

On this screen, the data entry fields are the same as those for creating a channel. See “Create channel” on page 81.

Upon exiting this screen, the program returns to the Select Channel to Modify screen.

Delete channel

Choice 9 on the Configuration Menu allows an operator to delete existing channel definitions.

To delete a channel, two screens are involved. The first allows the operator to select the desired channel from a list of defined channel. The selected channel definition is displayed on the second screen. On this screen, the operator is asked to verify the delete request. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                             ** Select Channel to Delete**

Channel Name            TYPE           Status         PID       MSN             Trace
================================================================================================
SYS1.TO.MVS1            SENDER         DISABLED       0000000   0000416         OFF
MVS1.TO.SYS1            REQUESTER      DISABLED       0000000   0004077         OFF
VAXOUT                  RECEIVER       IDLE           0014209   0000011         OFF
VAXINP                  SERVER         DISABLED       0014209   0000031         OFF
SYS1.TO.MVS1            SERVER         DISABLED       0000000   0000001         OFF
MVS1.TO.SYS1            RECEIVER       DISABLED       0000000   0001715         OFF
SYS1.TO.VSE2            SENDER         DISABLED       0000000   0000041         ON
VSE2.TO.SYS1            REQUESTER      DISABLED       0000000   0000021         ON
SYS1.TCPIP.TEXT.1       RECEIVER       DISABLED       0000000   0000257         ON




J       - Down                  K         -   Up                  <return>   - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X     - Exit
```

Figure 55. Select channel to delete

On this screen, each line displays the name and type of a defined channel. The operator uses the cursor control keys to highlight the desired channel and then presses <**Return**> to select the channel. Press <**Ctrl-X**> to return to the Configuration Menu. The selected channel is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                                ** Delete Channel **

      Channel Name: junk.channel                      MSN: 1
        Queue Name: Transmit_Queue
              Type: 1      1 = Sender 2 = Server 3 = Receiver 4 = Requester

* * * M A X I M U M S * * *                    * * * T I M E R S * * *
       Retry Count: 0                          Disconnect: 300
     MSN Wrap Count: 999999                     Reconnect: 30
   Checkpoint Count: 10                       Line Check: 20
      Message Size: 2048


   Transport Protocol: 0          0 = LU6.2      1 = TCP/IP


        Is this the Channel you wish to delete? [Y/N]: Y


<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 56. Delete channel

This screen displays the parameters of the channel definition which has been selected for deletion and prompts the operator to verify the delete request.

The operator responds:

Y to delete the displayed channel

N to abort the delete request

Upon exiting this screen, the program returns to the Select Channel to Delete screen.

Display channel

Choice 10 on the Configuration Menu allows an operator to view existing channel definitions.

To display a channel, three screens are involved. The first allows the operator to select the desired channel from a list of defined channels. The selected channel definition is displayed on the second screen. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                            ** Select Channel to Display **

Channel Name            TYPE           Status         PID       MSN            Trace
================================================================================
SYS1.TO.MVS1            SENDER         DISABLED       0000000   0000416        OFF
MVS1.TO.SYS1            REQUESTER      DISABLED       0000000   0004077        OFF
VAXOUT                  RECEIVER       IDLE           0014209   0000011        OFF
VAXINP                  SERVER         DISABLED       0014209   0000031        OFF
SYS1.TO.MVS1            SERVER         DISABLED       0000000   0000001        OFF
MVS1.TO.SYS1            RECEIVER       DISABLED       0000000   0001715        OFF
SYS1.TO.VSE2            SENDER         DISABLED       0000000   0000041        ON
VSE2.TO.SYS1            REQUESTER      DISABLED       0000000   0000021        ON
SYS1.TCPIP.TEXT.1       RECEIVER       DISABLED       0000000   0000257        ON




J       - Down                  K         -   Up                  <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 57. Select channel to display

On this screen, each line displays the name and type of a defined channel. Note that there may be situations in which the Status field displayed for a particular channel is inaccurate on this display. To get an accurate display of Status for all channels, use the Monitor Channel function (see “Monitor channel” on page 98). The operator uses the cursor control keys to highlight the desired channel and then presses <**Return**> to select the channel. Press <**Ctrl-X**> to return to the Configuration Menu.

The selected channel is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                                ** Display Channel **

      Channel Name: SYS1.TO.MVS1                      MSN: 1
        Queue Name: Transmit_Queue
              Type: 1       1 = Sender 2 = Server 3 = Receiver 4 = Requester

* * * M A X I M U M S * * *                    * * * T I M E R S * * *
       Retry Count: 1000                       Disconnect: 300
     MSN Wrap Count: 999999                     Reconnect: 30
   Checkpoint Count: 10                        Line Check: 20
      Message Size: 2048


   Transport Protocol: 0          0 = LU6.2      1 = TCP/IP

                      * * * Press any key to continue * * *
```

Figure 58. Display channel

This is a display-only screen.

The operator may press any key to display the second screen showing the transport layer parameters for either LU 6.2 or TCP/IP:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Display Channel (LU 6.2 Parameters)**

    Channel Name: SYS1.TO.MVS1                    Type: 1 SENDER

    Symbolic Destination Name:
                   Mode Name:      DFHLU62
             Source LU Name:       LWLU
            Partner LU Name:       MVSLU
    Transaction Program Name:      MVSRCV









                             *** Press any key to continue ***
```

Figure 59. Display channel - LU 6.2 parameters (SENDER channel)

This is a display only screen.

The operator may press any key to return to the Select Channel to Display screen:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Display Channel (TCP/IP Parameters)**

          Channel Name: SYS1.TO.MVS1                 Type: 1 SENDER


        Remote Hostname: MVS

          Service Name: SYS1.TO.MVS









                             *** Press any key to continue ***
```

Figure 60. Display channel (TCP/IP parameters)

This is a display only screen.

The operator may press any key to return to the Select Channel to Display screen.

Operation functions

Selecting option 2 (Operation) from the Main Menu causes MQM to display the following sub-menu screen:

```
                          IBM MQSeries for UnixWare Version 1
                                ** Operation Menu **



                              Enter Choice:       2

                              1.   Enable/Disable Channel

                              2.   Start/Stop Channel Trace

                              3.   Terminate MCA

                              4.   Reset Message Sequence Number

                              5.   Purge Deleted Messages



<return> - Select Option     <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Go to previous menu
```

Figure 61. Operation menu

On this screen, choices correspond to available operator control functions.

Enable/Disable channel

Choice 1 on the Operation Menu allows an operator to open or close communications on an existing channel. The MCA (Message Channel Agent) servicing the selected channel must be in operation in order to perform this function.

Note: Enabling/Disabling a Channel is NOT the same as Starting/Stopping the MCA process. See “Starting the MCA” on page 104.

To accomplish this, two screens are involved. The first allows the operator to select the desired channel from a list of defined channels. The selected channel definition is displayed on the second screen and the operator is asked to verify the enable/disable request. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Select Channel to Enable/Disable **

Channel Name            TYPE           Status          PID       MSN             Trace
================================================================================
VAXOUT                  RECEIVER       IDLE            0014209   0000011         OFF
VAXINP                  SERVER         DISABLED        0014209   0000031         OFF









J       - Down                  K         -   Up                <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                CTRL-X    - Exit
```

Figure 62. Select channel to enable/disable

On this screen, the operator uses the cursor control keys to highlight the desired channel and presses <**Return**> to select the channel. The selected channel is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                             ** Enable/Disable Channel **


    Channel Name: VAXOUT                        MSN: 11
      Queue Name: VAXOUT
            Type: 3 RECEIVER

*** MAXIMUMS ***                              *** TIMERS ***
      Retry Count: 10                         Disconnect: 300
   MSN Wrap Count: 999999                      Reconnect: 30
 Checkpoint Count: 10                         Line Check: 20
     Message Size: 11412

     Transport Protocol: 0                0 = LU6.2 1 = TCP/IP

        Do you wish to disable this channel?[Y/N]: N



<return> - FIELD EXIT       <esc>  - Discard Field Changes        CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes      CTRL-W - Save Changes
```

Figure 63. Enable/disable channel

This screen displays the parameters of the channel definition which has been selected and prompts the operator to verify the enable/disable request.

If the channel is currently enabled, the operator is asked: **Do you wish to disable this channel?** If the channel is currently disabled, the operator is asked: **Do you wish to enable this channel?**

The operator responds:

Y to enable/disable communications on the displayed channel

N to cancel the enable/disable request

Upon exiting this screen, the program returns to the Select Channel to Enable/Disable screen.

Start/stop channel trace

Choice 2 on the Operation Menu allows an operator to start or stop a channel trace.

A channel trace causes all key events on the channel to deposit informational messages in the log file. The file may be examined to assist in trouble-shooting communications problems.

Two screens are involved. The first allows the operator to select the desired channel from a list of active channels. The selected channel definition is displayed on the second screen and the operator is asked to confirm the request. If the channel is not being traced, the question is whether to start the trace. If the channel is being traced, the question is whether to stop the trace.

The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                        ** Select Channel to Start/Stop Trace **

Channel Name            TYPE            Status          PID       MSN            Trace
================================================================================
VAXOUT                  RECEIVER        IDLE            0014209   0000011        OFF
VAXINP                  SERVER          DISABLED        0014209   0000031        OFF









J       - Down                  K         -   Up                  <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 64. Start/stop trace

On this screen, the operator uses the cursor control keys to highlight the desired channel and then presses <**Return**> to select the channel. The selected channel is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                            ** Start/Stop Channel Trace **

     Channel Name: VAXOUT                             MSN:11
      Queue Name: VAXOUT
            Type: 3 RECEIVER

*** MAXIMUMS ***                              *** TIMERS ***
      Retry Count: 10                         Disconnect: 300
   MSN Wrap Count: 999999                      Reconnect: 30
 Checkpoint Count: 10                         Line Check: 20
     Message Size: 11412


   Transport Protocol: 0              0 = LU6.2 1 = TCP/IP


        Do you wish to start channel trace?[Y/N]: N


<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 65. Start/stop channel trace

The operator responds:

Y to start/stop the requested trace

N to cancel the start/stop trace request

Upon exiting, the program returns to the Select Channel to Start/Stop Trace screen.

Terminate MCA

Choice 3 on the Operation Menu allows an operator to terminate an MCA.

To accomplish this, two screens are involved. The first allows the operator to select the desired channel from a list of defined channels. The selected Process ID (PID) is displayed on the second screen and the operator is asked to confirm. Note that communications across all channels associated with that MCA (PID) will be terminated. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                          ** Select Channel(s) MCA to Stop **

Channel Name          TYPE          Status         PID       MSN             Trace
================================================================================
VAXOUT                RECEIVER      IDLE           0014209   0000011         OFF
VAXINP                SERVER        DISABLED       0014209   0000031         OFF









J       - Down                K         -   Up                <return>  - Select
CTRL-F  - PgDn                CTRL-B    - PgUp                CTRL-X    - Exit
```

Figure 66. Select channel to terminate MCA

On this screen, the operator uses the cursor control keys to highlight the desired channel and then presses <**Return**> to select the channel.

Note: Stopping any channel started by a specific MCA will result in terminating all channels associated with that MCA. The selected MCA is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                                  ** Terminate MCA **


Terminate MCA with PID: 14209 N









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 67. Terminate MCA

On this screen, the operator keys a "**Y**" to indicate that the MCA should be terminated, and then presses <**Return**>. The default action is "**N**", the MCA will continue to run. Upon exiting this screen, the program returns to the Select Channel to Terminate MCA screen.

Reset message sequence number (MSN)

Choice 4 on the Operation Menu allows an operator to reset the message sequence numbers on an existing channel. You may want to reset the message sequence numbers if your local and remote channel definitions get out of synchronization. A channel must be disabled in order for this operation to be allowed.

To accomplish this, two screens are involved. The first allows the operator to select the desired channel from a list of defined channels. The selected channel is displayed on the second screen and the operator is asked to enter the new message sequence number. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                          ** Select Channel to Modify MSN **

Channel Name            TYPE           Status         PID       MSN             Trace
================================================================================
SYS1.TO.MVS1            SENDER         DISABLED       0000000   0000416         OFF
MVS1.TO.SYS1            REQUESTER      DISABLED       0000000   0004077         OFF
VAXOUT                  RECEIVER       IDLE           0014209   0000011         OFF
VAXINP                  SERVER         DISABLED       0014209   0000031         OFF
SYS1.TO.MVS1            SERVER         DISABLED       0000000   0000001         OFF
MVS1.TO.SYS1            RECEIVER       DISABLED       0000000   0001715         OFF
SYS1.TO.VSE2            SENDER         DISABLED       0000000   0000041         ON
VSE2.TO.SYS1            REQUESTER      DISABLED       0000000   0000021         ON
SYS.TCPIP.TEXT.1        RECEIVER       DISABLED       0000000   0000257         ON




J       - Down                  K         -   Up                  <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 68. Select channel to modify MSN

On this screen, the operator uses the cursor control keys to highlight the desired channel and presses <**Return**> to select the channel. The selected channel is then displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                               ** Modify Channel MSN **


    Channel Name: MVS1.TO.SYS1              Type:4              MSN:1









<return> - FIELD EXIT       <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE        CTRL-X - Exit discarding changes        CTRL-W - Save Changes
```

Figure 69. Reset MSN

This screen displays the channel definition which has been selected and allows the operator to enter the new message sequence number.

The display fields are:

Channel Name: Filled in from previous screen. Cannot be modified.

Type: Filled in from previous screen. Cannot be modified.

MSN: New Message Sequence Number (sequence number of the next message on this channel). Must be greater than or equal to 1.

Upon exiting this screen, the program returns to the Select Channel to Modify MSN screen.

Notes:

1. The action described above does not cause a reset request to be sent to the remote queue manager; the user must reset the message sequence number there as well if required. However, if a reset request is received from a remote queue manager, the message sequence number at this queue manager is reset.
2. If the specified Message Sequence Number is greater than the wrap value, the effect is the same as setting the Message Sequence Number to 1.
3. Care must be exercised in the use of this function. If incorrectly used, it is possible for messages to be lost or duplicated. If the message sequence number is the same at both ends of the channel, then when the channel is started, the MCAs will assume that all previously-sent messages have been correctly received. If the message sequence number at the sending end is one less than that at the receiving end, it will be assumed that the last message transmitted was not successfully received.
4. If the operating system fails whilst messages are being sent, then, after the system has been rebooted, the Message Number for any channel which was active when the failure occurred may not be correctly reflected in the MQM panels. This will be reset correctly once the channel is restarted.

Purge deleted messages

Choice 5 on the Operation Menu allows an operator to purge messages physically which have already been logically deleted from a specified queue.

Note: Messages which have been read are not actually deleted; instead, they are marked. Therefore, the queue can become quite large, unless it is purged automatically or by the operator. Purging actually removes old messages.

To purge messages, the operator is first presented with a list of defined queues as shown on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                             ** Purge Deleted Messages **

Queue                                Type      USERS     LWRIT     DEPTH     G P
=============================================================================================
Local_Queue                          LOCAL     00000     00000     00000     A A
Transmit_Queue                       TRANSMIT  00000     00000     00000     A A
DEADLETTER                           LOCAL     00000     00000     00000     A A
SYS1.MVS.LOCALQ1                     LOCAL     00000     00000     00000     A A
VAXINP                               TRANSMIT  00019     00000     00000     A A
VAXOUT                               LOCAL     00000     00000     00000     A A
VAXQMGR                              TRANSMIT  00000     00000     00000     A A
VMS_Queue_Manager                    TRANSMIT  00000     00020     00020     A A




J       - Down              K         -   Up              <return>  - Select
CTRL-F  - PgDn              CTRL-B    - PgUp              CTRL-X    - Exit
```

Figure 70. Select queue to purge

On this screen, the operator uses the cursor control keys to highlight the desired queue and then presses <**Return**> to select the queue. All previously deleted messages are then purged (physically deleted) from the selected queue.

The purge operation may only be performed on queues not currently being accessed. That is, the selected queue must have USERS=0. Additionally, channels that have opened queues may need to be terminated before the purge operation can be performed. The presence of unpurged messages on a queue can affect performance.

Messages are purged by copying non-deleted (that is, active) messages to a new queue file. The old queue file is deleted, and then the new queue file name is changed to match the old queue file name. As a result, the new queue file shall be owned by the person running `MQM`.

Upon exiting this screen, the program returns to the Operation Menu.

Monitoring functions

Selecting option 3 (Monitoring) from the Main Menu causes MQM to display the following sub-menu screen:

```
                          IBM MQSeries for UnixWare Version 1
                                  ** Monitor Menu **



                              Enter Choice:       1

                              1.   Monitor Queue

                              2.   Monitor Channel









<return> - Select Option     <esc>  - Discard Field Changes          CTRL-D - Erase Field
<BKSP>   - BACKSPACE         CTRL-X - Go to previous menu
```

Figure 71. Monitor menu

On this screen, choices correspond to available system monitor functions.

Monitor queues

Choice 1 on the Monitor Menu allows an operator to monitor the current status of all existing local queues. The monitor screen displayed is:

```
                              IBM MQSeries for UnixWare Version 1
                                  ** Monitor Local Queues **

Queue                                 Type      USERS     LWRIT     DEPTH     G P
================================================================================
Local_Queue                           LOCAL     00000     00000     00000     A A
Transmit_Queue                        TRANSMIT  00000     00100     00100     A A
DEADLETTER                            LOCAL     00000     00000     00000     A A
Example_Local_Queue                   LOCAL     00000     00000     00000     A A
VAXINP                                TRANSMIT  00031     00000     00000     A A
VAXOUT                                LOCAL     00000     00000     00000     A A
VAXQMGR                               TRANSMIT  00000     00000     00000     A A
VMS_Queue_Manager                     TRANSMIT  00000     00100     00100     A A









J       - Down                  K         -   Up                  <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                  CTRL-X    - Exit
```

Figure 72. Monitor local queues

This screen displays the current status of all local queues. The displayed fields are:

Queue: Queue Name.

Type: Type of Queue, local or transmit.

USERS: Number of user applications currently connected to the queue.

LWRIT: Last written message to queue. This is a sequential count indicating how many messages have been written to the queue since it was last purged. The queue disk file contains this many physical records.

DEPTH: The number of unread messages currently on queue.

Note: The difference between LWRIT and DEPTH (LWRIT-DEPTH) is the number of logically deleted messages still on disk. This indicates the amount of disk space which can be gained by purging the queue.

G: Get operation allowed/inhibited.

P: Put operation allowed/inhibited.

For both G and P, the displayed value is:

A for Active/Allowed.

I for Inactive/Inhibited.

Upon exiting this screen, the program returns to the Monitor Menu.

Monitor channel

Choice 2 on the Monitor Menu allows an operator to monitor the current status of all communications channels.

The current status for all channels is displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1
                                ** Monitor Channels **


Channel Name           TYPE           Status         PID       MSN            Trace
================================================================================
VAXOUT                 RECEIVER       IDLE           0014209   0000011        OFF
VAXINP                 SERVER         DISABLED       0014209   0000031        OFF









J       - Down                 K         -   Up                 <return>  - Select
CTRL-F  - PgDn                 CTRL-B    - PgUp                 CTRL-X    - Exit
```

Figure 73. Monitor channels

This screen displays the current status of all channels. The displayed fields are:

Channel Name: Channel name.

TYPE: Type of channel.

Status: ACTIVE, IDLE, or DISABLED. Depending on the TYPE of channel, each Status type is interpreted according to Table 18, on page 99.

PID: The Process ID of the MCA controlling this channel.

MSN: Message Sequence Number.

Trace: OFF or ON.

Upon exiting this screen, the program returns to the Monitor Menu.

Table 18. Channel status descriptions

ACTIVE	
SENDER	The Channel is trying to (re)establish communication with the remote receiver (based upon the channels retry and reconnect interval).
	The Channel is negotiating communications values with its partner.
	The Channel is sending messages.
	The Channel is waiting for the Disconnect interval to expire.
SERVER	The Channel is negotiating communications values with its partner.
	The Channel is sending messages.
	The Channel is waiting for the Disconnect interval to expire.
RECEIVER	The Channel is negotiating communications values with its partner.
	The Channel is receiving messages from the remote channel.
	The Channel is waiting for the remote channel to Disconnect (close).
REQUESTER	The Channel is trying to (re)establish communication with the remote server (based upon the channels retry and reconnect interval).
	The Channel is negotiating communications values with its partner.
	The Channel is receiving messages from the remote channel.
	The Channel is waiting for the remote channel to Disconnect (close).
IDLE	
SENDER	The Channel is waiting for messages to be put on the transmission Queue. When a message arrives on the transmission Queue, the channel will start.
	The Channel is waiting for the reconnect interval to expire, so it can attempt to contact the remote system again.
SERVER	The Channel is waiting to be contacted by a REQUESTER channel.
RECEIVER	The Channel is waiting to be contacted by a SENDER channel.
REQUESTER	The Channel is waiting for the reconnect interval to expire, so it can attempt to contact the remote system again.

Table 18. Channel status descriptions (continued)

DISABLED	
SENDER	The Channel has been disabled by the MQM disable channel function and when appropriate all of the retry attempts have been performed.
	A communications error has occurred (or the channel has been instructed to stop by the remote channel due to an error in the remote queue manager) and (when appropriate) all off the retry attempts have been performed.
SERVER	The Channel has been disabled by the MQM disable channel function.
RECEIVER	The Channel has been disabled by the MQM disable channel function.
REQUESTER	The Channel has been disabled by the MQM disable channel function.
	A communications error has occurred, and all of the retry attempts have been performed.
	The Channel has been disabled as the remote channel has sent all of the available messages and disconnected (closed).

Browse function

Selecting option 4 (Browse QUEUE Records) from the Main Menu takes the operator directly to a function with no intervening sub-menus.

To browse the contents of a queue, two screens are involved. The first allows the operator to select the desired queue from a list of defined queues. The records (messages) on the selected queue are displayed on the second screen. The first screen displayed is:

```
                          IBM MQSeries for UnixWare Version 1
                             ** Select Queue To Browse **



Queue                           Type      USERS     LWRIT     DEPTH     G P
================================================================================
Local_Queue                     LOCAL     00000     00000     00000     A A
Transmit_Queue                  TRANSMIT  00000     00100     00100     A A
DEADLETTER                      LOCAL     00000     00000     00000     A A
Example_Local_Queue             LOCAL     00000     00000     00000     A A
VAXINP                          TRANSMIT  00031     00000     00000     A A
VAXOUT                          LOCAL     00000     00000     00000     A A
VAXQMGR                         TRANSMIT  00000     00000     00000     A A
VMS_Queue_Manager               TRANSMIT  00000     00100     00100     A A



J       - Down                  K         -   Up                <return>  - Select
CTRL-F  - PgDn                  CTRL-B    - PgUp                CTRL-X    - Exit
```

Figure 74. Select queue to browse

On this screen, the operator uses the cursor control keys to highlight the desired queue and then presses <**Return**> to select the queue. All records on the queue are displayable, even if they are logically deleted. The first record on the selected queue is displayed on the following screen:

```
                          IBM MQSeries for UnixWare Version 1

Queue:VMS_Queue_Manager                                  Rec: 00001 Page: 01/04

EZQH1.01A.............15-A       455A5148  312E3031  41000000  00000400  00000000  31352D41
pr-93   18:38:34.+.........      70722D39  33202031  383A3338  3A333400  2BCDE3EA  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
........                       00000000  00000000  20202020  20202020  20202020  20202020
                               20202020  20202020  20202020  20202020  20202020  20202020
VMS_Local_Queue                564D535F  4C6F6361  6C5F5175  65756520  20202020  20202020
                               20202020  20202020  20202020  20202020  20202020  20202020
MD ......................      4D442020  00000001  00000000  00000008  FFFFFFFF  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000
.........................      00000000  00000000  00000000  00000000  00000000  00000000

<+> - Record Forward           CTRL-F - Page Forward                CTRL-X - Exit
<-> - Record Backward          CTRL-B - Page Backward
```

Figure 75. Browse queue record

This screen initially displays the first record on the selected queue. The available action keys listed at the bottom of the screen allow the operator to browse forward and backward through the contents of the queue.

Upon exiting this screen, the program returns to the Select Queue to Browse screen.

The MCAMD process

The MCAMD (Message Channel Agent Maintenance Daemon) is a daemon process required by the MCA(s) (Message Channel Agent(s)). It runs on the communication server with and serving the MCA(s). The MCAMD provides a centralized Channel Database service allowing MCA(s) and the MQM interface to access and modify the Channel Database. The MCAMD must be started prior to running an instance of MCA.

Starting the MCAMD

The MCAMD is started using the following command line:

```
[nohup] mcamd [-u user_id] [-g group_id] [&]
```

in which:

Table 19. MCAMD options

[....]	indicates parameter is optional
nohup	(optional) used to prevent the MCAMD from exiting due to sighup signal, if the daemon is started interactively from a terminal window and the window terminates or is exited.
-u	(optional) switch identifying the following argument as a user identification.
user_id	(optional) a numeric user id as defined in the password file /etc/passwd.
-g	(optional) switch identifying the following argument as a group identification.
group_id	(optional) a numeric group id as defined in the group file /etc/group.
&	(optional, but recommended) allows MCAMD to run in background mode.

The options available permit the system administrator to set the process user and group id to values which allow access by those persons responsible for administering the MQSeries System Product (that is, starting and stopping MCA processes).

It is recommended that the MCAMD process be invoked by the UnixWare “init” process by adding an entry to the file /etc/inittab in the following format:

```
mcad:2:respawn:/usr/mqi/bin/mcamd -u user_id -g group_id > /dev/console 2>&1
```

Refer to “Verifying the MQSeries System installation” on page 10, for information regarding the setting of the NLSPATH environment variable prior to starting the MCAMD process.

The above-mentioned entry ensures that the MCAMD process will be running at all times when the UnixWare operating system is in the multi-user mode, run level 2. To stop the MCAMD, refer to “Stopping the MCAMD” on page 103.

As an alternative, the MCAMD may be started from a shell prompt as:

```
mcamd [-u user_id] [-g group_id] &
```

However, to use -u and/or -g options, root authority is needed. If you wish to start the MCAMD and MCA(s) within a shell script, you must allow the MCAMD to complete its initialization prior to starting the first MCA. This can be accomplished by the following lines:

```
nohup mcamd [-u user_id] [-g group_id] &
sleep 5
mca -m queue_manager_name -f filename -c channel_name &
mca -m queue_manager_name -f filename -c channel_name &
```

For further information on the MCA process, please refer to “Starting the MCA” on page 104 and “MCA shutdown” on page 105.

The MCAMD, the MCA(s) and the MQM utilities communicate through the use of sockets, both UNIX and INTERNET, and shared memory segments. These mechanisms are automatically maintained by both processes.

Stopping the MCAMD

If the “init” process is not being used to automatically start the MCAMD, then the MCAMD may simply be stopped with the command:

```
kill <pid>
```

where <pid> is the process id of the MCAMD, displayable with the command:

```
ps -aef | grep mcamd
```

If the “init” process is being used to automatically start the MCAMD, killing the MCAMD process will not work. The “init” process will detect that the MCAMD has been killed and restart it. In this case, the MCAMD may be stopped by changing the word respawn to off and typing the command:

```
/etc/init Q
```

which signals the init process to recognize the modification to the /etc/inittab file.

Note: Any MCA(s) which were started with a defunct MCAMD cannot be controlled by a new MCAMD, such as one which is respawned. To establish control, you must kill the MCAs from the command line and restart them.

If for some reason the MCAMD is stopped (killed), the MCA processes will continue to run without a problem with the following conditions:

- The operator interface through the MQM utility will not function with regard to any MCA.
- Checkpointing of the Channel Database will not occur. This is non-critical in that the only side effect is the increased time to verify the Message Sequence Number on a restart of the MQSeries System product.
- You will be unable to start any new MCA processes.

Warning: It is highly recommended that the MCAMD and the MCA processes not be killed or stopped with a signal SIGKILL (9) unless absolutely necessary. This defeats all the orderly shutdown mechanisms imbedded in these processes. If you must use a SIGKILL signal to shutdown the MCAMD, you must verify that all shared memory segments used by the MCAMD and the MCA(s) are deleted prior to restarting any of these processes.

MCAMD error logging

The MCAMD performs error logging as do all other processes containing the MQSeries System components (that is, MCA and user applications using the MQI API). If the error log grows to the size that requires a user to create a new log, all processes using that log must be stopped. Simply deleting the error log will not free up the disk space used by the log file.

The MCAMD process may be restarted quickly by performing the following actions in the order listed:

- Stop all MCA processes (refer to “MCA shutdown” on page 105).
- Delete or rename the log file.
- Issue the command; `kill -TERM pid`, where `pid` is the process id of the MCAMD.
- Start the MCAMD process.
- Start all MCA processes.

It is obvious that this is a non-trivial operation which disrupts the operation of the MQSeries System. Users can avoid or minimize the problem by using the trace facility of the MCA only when necessary in order to minimize log-writing and to ensure that sufficient disk space is available to store log files over a period of time.

Starting the MCA

MCA (Message Channel Agent) is the communications engine for the MQSeries System. It runs on the communications server and connects the UnixWare MQSeries system to remote MQSeries Systems. Once MCA is started, its operation may be controlled from the MQM screens. However, it must first be started manually.

If needed, you can start more than one MCA process for supporting multiple channels. Each MCA instance may service a maximum of 20 channels. A maximum of nine MCA instances may be started on one IBM MQSeries for UnixWare installation.

Each required MCA instance is started using the following command line at the shell prompt:

```
[nohup] mca -m queue_mgr_name [-f filename] [-c chan_name \ [-c chan_name...]] [&]
```

in which:

[.....]	indicates parameter is optional.
nohup	(optional) used to prevent the MCA from exiting due to sighup signal, if the MCA is started interactively from a terminal window and the window terminates or is exited.
-m	(required) switch identifying the following argument as the queue manager name.
queue_mgr_name	(required) the name of the queue manager supported by this MCA. This name must match the name of the local queue manager, obtainable from the display queue manager panel.
-c	(optional, but at least one -c or -f must be present) switch identifying the following argument as a channel name.
chan_name	(optional) the name of a channel to be serviced by this MCA. This name must match the name of one of the locally defined channels.
-f	(optional, but at least one -c or -f must be present) switch identifying the following argument, as file name.
filename	(optional) the name of a source file containing a list of channel_names to be serviced by this MCA. Each name in the file must match the name of a locally defined channel.
&	(optional, but preferred) allows MCA to run as a background process.

Examples:

To invoke MCA to service two explicit channels named *channel_1* and *channel_2*, enter:

```
mca -m queue_manager_name -c channel_1 -c channel_2 &
```

To invoke MCA to service a list of channels contained in file *chanlist*, enter:

```
mca -m queue_manager_name -f chanlist &
```

In the latter case, the contents of the file *chanlist* would be (for example):

```
channel_1
channel_2
channel_3
channel_4
<End-of-File>
```

Note: The format of the channel list file is very simple. There is only one channel name per line and there can be no leading spaces.

In response to this command, MCA will begin execution. All other MCA functions may be controlled from the MQM display screens.

Note: Prior to running the MCA, you must insure the prerequisite communications hardware and software are properly installed and configured (See Chapter 2, “Installation” on page 5) to support the lower level transport layer connection to the remote MQSeries System.

MCA shutdown

The MQM Operation Menu, option 3, Terminate MCA, allows you to shutdown an instance of the MCA. The screen lists all the channels currently supported by MCA processes. If you started an MCA supporting more than one channel you will stop all the channels supported by this MCA process when you chose one from the list presented. The Process IDs of the MCAs are listed alongside the channels. This is to enable you to determine which channels will be affected by terminating a given MCA.

As described above, you may elect to send the MCA process a signal. The signals SIGINT, SIGQUIT and SIGTERM are processed by the MCA resulting in an orderly shutdown. Shutdown in this manner removes the visual feedback of the MQM from the decision making process. Therefore, care should be exercised.

If the MCA process you wish to terminate is supporting multiple channels, all channels will attempt to do an orderly shutdown. RECEIVER/REQUESTER channels by definition can only request a close of the channel, they are not capable of issuing a close channel command. This request is returned in a status message sent to a SENDER/SERVER channel. Status messages are returned on receipt of a data message or as a result of line check message. The RECEIVER/REQUESTER channels will NOT shutdown until they are able to request a close and a SENDER/SERVER return a close channel. Therefore, if your channels are unlikely to transport messages at a regular interval which you determine is a reasonable latency between the time you perform the action required to shutdown an MCA and the actual termination of the process, it is important that the line check timer be set at that interval. This latency will be equal to the longest interval described.

Note: If EXPRESS software is being used (SNA/LU6.2 transport links), it is important that EXPRESS is shut down while the MCA that is using it is still executing.

If Terminate MCA Command Fails

If the Terminate MCA command should fail to shutdown the MCA, use the command:

```
kill -9 <pid>
```

where <pid> is the MCA process id displayed on the Terminate MCA screen or found by using the command:

```
ps -aef | grep mca
```

Viewing error logs with OS utilities

MQSeries System error messages and trace messages (if requested) are logged to an error file named:

`/var/mqi/log/mqi`*dd*`.log`

where *dd* is the day of the month.

If everything is operating normally, log files will be quite small. If many errors are being encountered, the files can be large and can contain very useful information. They may be particularly useful when first attempting to establish a communications link to a remote system. They should also be checked when any unusual events are observed.

The error messages in the log file are in plain text format so that the file can be examined using any editor or various commonly available utilities.

Notes:

1. A new log file is created every day. These should be examined periodically and old unwanted logs should be deleted to preserve disk space.
2. A log file that already exists will be appended to and not overwritten.

Chapter 6. Application programming interface

The MQSeries System application programming interface implements the IBM Message Queue Interface (MQI). This simple set of calls provides a way for applications to easily send and receive messages between workstations on the same LAN, and to exchange messages with other MQSeries Systems such as VAX, TANDEM, AS/400, etc.

The applications programmer/analyst/designer should read earlier chapters of this document for an overall understanding of the MQSeries System. The reference documents listed in the appendix will provide further background information.

In addition to these sources, this chapter provides:

- General information regarding the MQI
- Design guidelines for applications wishing to use the MQI
- Detailed reference for each individual MQI function
- Description of key MQI data structures
- Completion codes and reason codes returned by MQI functions.

Working with the MQI

The MQI is responsible for handling user application requests to read and write from the queuing system, and for arbitrating among multiple requests to the same queue.

On the MQSeries System, the MQI is built around the standard C language function call interface which allows a fixed number of arguments.

MQI calls & sequence of operations

The MQI calls supported by IBM MQSeries for UnixWare are:

MQCONN	Connects the application to the MQSeries System Queue Manager
MQOPEN	Opens access to a specific queue
MQGET	Reads a message from a specified queue
MQPUT	Writes a message to a specified queue
MQPUT1	Opens a queue, writes one message, and closes the queue
MQINQ	Inquires about queue status information
MQCLOSE	Closes access to a specific queue
MQDISC	Disconnects the application from the MQSeries System Queue Manager

These calls are described in detail in “MQI call reference” on page 110. It is also important to understand the data structures required by the interface -- especially as part of the `MQGET` and `MQPUT` calls. The primary structures are:

MQOD	MQ Object Descriptor
MQMD	MQ Message Descriptor
MQPMO	MQPut Message Options
MQGMO	MQGet Message Options
MQDLH	MQ Dead-Letter Header

The use of these data structures is described along with the MQI call descriptions in “MQI call reference” on page 110. The structures are described independently in “MQI data types and structures” on page 129.

The sequence of MQI operations performed by an application is very similar to the sequence used for any familiar record-oriented I/O subsystem. That is, just as one must OPEN and CLOSE a disk file, one must connect to and MQOPEN a queue before accessing it, and must MQCLOSE and then disconnect at the completion of processing. Within the application, the user requirements will determine the sequence of MQGET and MQPUT operations.

Sample source code provided

Three sample programs are provided with IBM MQSeries for UnixWare. They are zmqwrite, zmqread, and zmqecho. The source code for these programs can be found in Appendix B, “Sample source listings” on page 167, or they can be listed directly from the distribution files.

Within the source code for zmqwrite, zmqread, and zmqecho, the user may find examples which illustrate the use of each of the MQI calls.

In addition to the three sample programs, several C language header files are provided with the distribution in the INCLUDE directory.

Compiling your application program

The MQI is provided in the form of an object library called libmqi.a, and a shared library called libmqi.so.

A typical UNIX application, foo.c, would be compiled with the following command:

```
cc -I install_dir/mqi/include -L install_dir/mqi/lib -lmqi -lnsl -lsocket foo.c
```

where: *install_dir* is the user-defined directory that was specified during installation.

We recommend that you do not turn on the optimization flag.

Applications not written in C

For IBM MQSeries for UnixWare, C is the language in which the MQI is written. Applications written in C have been thoroughly tested with the MQI. Include modules written in C are provided for MQI data structures. Sample programs are provided in C.

C is clearly the language of choice for development of MQSeries applications on UNIX. However, for a variety of reasons, some users will want to write in another programming language.

In these cases, the *customer* must meet the interface requirements of the C language interface. There are no sample programs and no includable header files provided in any other language.

Nevertheless, any programming language which can call C routines should be able to be used in one of two manners:

- Call the MQI directly from another language. This usage requires that all interface parameters match up identically at the binary level. With some languages this may present a problem.

or

- Within the application, call an application subroutine written in C. From this C language subroutine issue the MQI calls. In this manner, there should be no problem with data alignment.

Application design guidelines

The hidden network

One of the key benefits provided by the MQI is the ability for a distributed application to be developed which is totally independent of the underlying network. This network independence means there is no need for an application to be aware of *either*:

- The lower levels of the communication protocol(s)

or

- The physical location of other applications on the network.

In order to take full advantage of this network independence, the queue names used by the application must be chosen properly.

In particular, it is recommended that application programs use only a *single logical name* to refer to each MQSeries System queue. For the MQI calls, this means only the *Queue_Name* field is used to identify queues. The use of the queue's fully qualified name (which includes both the *Queue_Name* field and the *Queue_Manager_Name* field) is not recommended (except when replying to a request message; in this case, the fully qualified name of the reply-to queue is presented to the application with the request message).

The reasoning behind this recommendation parallels the logical naming used in other I/O subsystems. When dealing with disk subsystems, no application hard-codes the device name and path name for a file. This would cause havoc for the application when normal system management functions relocate a file.

The same is true when addressing MQSeries System queues. Since the *Queue_Manager_Name* is typically associated with a particular system, its use implies knowledge of the physical network. This can place restrictions on any future modifications to the network and increase the probability that network changes will require changes to the source code of applications.

Note: The use of the Queue_Name field as the only logical queue name is strongly recommended. This usage maximizes application flexibility and network independence. The mapping of the queue name in this form to the proper network destination then becomes a configuration issue to be handled by the MQSeries System administrator.

This recommended usage should be reflected in the list of queue names defined by the system designer (as described in Chapter 3, "Planning" on page 21).

Syncpoint considerations

The IBM MQSeries for UnixWare provides no explicit support for coordination of queue functions with logical units of work. However, since all queues are disk resident files, they may be managed by any transaction monitor or syncpoint software available on the system.

Triggering

Triggering is the ability for the MQSeries System software to notify, or awaken, an application when a message arrives on a particular local queue.

The IBM MQSeries for UnixWare provides only a *pseudo-triggering* capability.

This pseudo-triggering feature is available via the `MQGET` call using the `MQGMO` structure and the `MQGMO_WAIT` options. When this is specified, the action taken by the MQSeries System is as follows:

- If the target queue is non-empty when the `MQGET` is issued, then the first message is returned normally.
- If the target queue is empty when the `MQGET` is initially issued, MQSeries System will *not return control to the application, but* will periodically poll the queue for a message arrival. In this case, control is returned to the application when either a message arrives or the time specified in WaitInterval expires. The frequency with which the queue is polled is specified in Queue Manager.

If the application wishes to implement a different form of control, then the logic must be incorporated in the application itself.

MQI call reference

For each of the MQI functions, this section presents the detailed *call format*, *parameters*, and *guidelines* in the following format:

The API calls are described using the following conventions:

MQAPINAME - Call Name

```
MQAPINAME (Parameter 1, Parameter 2, Parameter 3, ..., Parameter N)
```

Description of how and when to use the API call.

Parameters

Parameter 1 - Parameter Type (see below)

Description of Parameter.

Instructions for using the parameter.

- PARAMETER_OPTION - description
- PARAMETER_OPTION_2 - description.

Guideline

Guidelines and tips for using the call.

Parameter types:

- **Input** - Parameter set by the application for use by the queue manager. A null byte is treated as a blank.
- **Output** - Parameter set by the queue manager for use by the application on return from the call. These parameters are always blank padded.
- **Input/Output** - Set by the application for use by the queue manager, and modified by the queue manager for use by the application on return from the call.

Note: Features of the full IBM MQI that are not supported in IBM MQSeries for UnixWare are so noted in the following sections.

MQCONN - Connect queue manager

```
MQCONN (Name, Hconn, CompCode, Reason)
```

The MQCONN call connects an application program to a queue manager. It provides a queue manager handle, which is used by the application on subsequent message-queuing calls.

Before any of the message-queuing services can be used, the application must establish a connection to a queue manager. The application does this by means of the MQCONN call.

The application provides the name of the queue manager required (Name), and receives in return a handle (Hconn) that represents the connection to that queue manager.

The returned handle is needed for all subsequent calls on that connection.

CompCode and Reason are returned parameters that indicate the success or failure of the call.

The application can connect either to a specified queue manager, or to the default queue manager.

The default queue manager is requested by specifying a name consisting entirely of blanks. The queue manager specified must be local to the application.

Parameters

Name (MQCHAR48) - input

Name of the queue manager.

The name specified must be the name of a local queue manager; if the name consists entirely of blanks, the name of the default queue manager is used.

The name must not contain leading or embedded blanks, but may contain trailing blanks; the first null character and characters following it are treated as blanks.

Hconn (PMQHCONN) - output

Connection handle.

This handle represents the connection to the queue manager. It must be specified on all subsequent message-queuing calls issued by the application. It ceases to be valid when the MQDISC call is issued, or when the application ends.

CompCode (PMQLONG) - output

Completion code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_WARNING - Warning (partial completion).
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report.

If CompCode is MQCC_WARNING:

- MQRC_ALREADY_CONNECTED - Application already connected.

If CompCode is MQCC_FAILED:

- MQRC_ACCESS_RESTRICTED - Queue manager in restricted access mode.
- MQRC_MAX_CONN_LIMIT_REACHED - Maximum number of connections reached.
- MQRC_NOT_AUTHORIZED - Not authorized for access.
- MQRC_Q_MGR_NAME_ERROR - Queue manager name not valid or not known.
- MQRC_Q_MGR_NOT_AVAILABLE - Queue manager not available for connection.
- MQRC_SECURITY_ERROR - Security error occurred.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.

See “MQI return codes” on page 144, for more details.

Guidelines

1. Only a local queue manager can be connected using this call; it is not possible to connect to a remote queue manager. Queues which belong to the connected queue manager appear to the application as local queues. Queues belonging to local queue managers other than the connected queue manager appear as remote queues. Queues belonging to remote queue managers also appear as remote queues.
2. After a failure of a queue manager, this call must be reissued. The application program can keep reissuing MQCONN calls until it finds that the queue manager has been restarted. If an application is not sure whether or not it is connected to the queue manager, it can safely reissue an MQCONN call. If it is already connected, the same handle is returned as was returned for the previous MQCONN call.
3. The MQDISC call is used to disconnect from the queue manager.

MQOPEN - open message queue

```
MQOPEN (Hconn, ObjDesc, Options, Hobj, CompCode, Reason)
```

The MQOPEN call establishes access to a queue object.

When a connection to the queue manager has been established, the application can open one or more queues for putting or getting messages. A queue is opened by means of the MQOPEN call.

The application specifies the queue to be opened (ObjDesc), and options (Options) that indicate whether the queue is opened for putting or getting messages.

The application receives in return a handle (Hobj) to the opened queue. The returned handle is used on subsequent calls to access the queue.

Parameters

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

ObjDesc (PMQOD) - input/output

Object descriptor.

This is the structure that identifies the object to be opened; see MQOD in “MQOD - MQ object descriptor structure” on page 131 for details.

Options (MQLONG) - input

Options that control the action of the MQOPEN call.

One or more of the following must be specified. If more than one is required, the values are added together.[1] Combinations that are not valid are noted; all other combinations are valid. Only options that are applicable to the type of object specified by ObjDesc are allowed.

The options for controlling the action of MQOPEN are as follows:

- MQOO_INPUT_SHARED - Open to get messages with shared access. The queue is opened for use with subsequent MQGET calls. The call can succeed if this queue is currently open, by this or another application, with MQOO_INPUT_SHARED, but fails if it is currently open with MQOO_INPUT_EXCLUSIVE. Only one of MQOO_INPUT_SHARED and MQOO_INPUT_EXCLUSIVE options can be specified.

Note: Please refer to "The MQSeries System's distributed architecture on UNIX" on page 3, for additional information regarding the usage of MQOO_INPUT_SHARED.

- MQOO_INPUT_EXCLUSIVE - Open to get messages with exclusive access. The queue is opened for use with subsequent MQGET calls. The call fails if this queue is currently open, by this or another application, for input of any type (MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE). Only one of MQOO_INPUT_SHARED and MQOO_INPUT_EXCLUSIVE options can be specified.
- MQOO_BROWSE - Open to browse messages with record locking (See "MQGMO - MQGet message options structure" on page 138 for usage details). The queue is opened for use with subsequent MQGET calls with the MQGMO_BROWSE_FIRST option. An MQOPEN call with the MQOO_BROWSE option establishes a browse cursor, and positions it logically before the first message on the queue.
- MQOO_OUTPUT - Open to put messages. The queue is opened for use with subsequent MQPUT calls.
- MQOO_INQUIRE - Open to inquire about object attributes. The queue is opened for use with subsequent MQINQ calls.

Table 20. Valid open options for each queue type

Option	Alias[a]	Local	Remote
MQOO_INPUT_SHARED	X	X	
MQOO_OUTPUT	X	X	X
MQOO_INQUIRE	X	X	X
MQOO_BROWSE	X	X	
MQOO_INPUT_EXCLUSIVE	X	X	

a. The validity of an alias depends on the validity of the queue to which the alias resolves.

If an alias queue is being opened for input (browse does not count as input), the test for exclusive use (or for whether another application has exclusive use) is against the base queue to which the alias queue resolves.

Hobj (PMQHOBJ) - output

Object handle.

This handle represents the access that has been established to the object. It must be specified on subsequent message-queuing calls that operate on the object. It ceases to be valid when the MQCLOSE call is issued, or when the application ends.

1. Do not add the same constant more than once.

CompCode (PMQLONG) - output

Completion Code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report.

If CompCode is MQCC_FAILED:

- MQRC_ALIAS_BASE_Q_TYPE_ERROR - Alias base queue not a valid type.
- MQRC_CONNECTION_BROKEN - Connection lost.
- MQRC_HANDLE_NOT_AVAILABLE - No more handles available.
- MQRC_HCONN_ERROR - Connection handle not valid.
- MQRC_NOT_AUTHORIZED - Not authorized for access.
- MQRC_OBJECT_IN_USE - Object already open with conflicting options.
- MQRC_OBJECT_TYPE_ERROR - Object type not valid.
- MQRC_OD_ERROR - Object descriptor structure not valid.
- MQRC_OPTION_NOT_VALID_FOR_TYPE - Options not valid for object type.
- MQRC_OPTIONS_ERROR - Options not valid or not consistent.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.
- MQRC_UNKNOWN_ALIAS_BASE_Q - Unknown alias base queue.
- MQRC_UNKNOWN_OBJECT_NAME - Unknown object name.
- MQRC_UNKNOWN_OBJECT_Q_MGR - Unknown object queue manager.
- MQRC_UNKNOWN_REMOTE_Q_MGR - Unknown remote queue manager.
- MQRC_XMIT_Q_TYPE_ERROR - Transmission queue not local.
- MQRC_XMIT_Q_USAGE_ERROR - Transmission queue with wrong usage.

See “MQI return codes” on page 144, for more details.

Guidelines

1. This call is used to open a queue in order to:
   - Get messages (using MQGET call).
   - Put messages (using the MQPUT call).
   - Inquire about the attributes of the queue (using the MQINQ call).
2. It is invalid for an application to directly open for output a local queue created with Usage specified as Transmission.
3. It is valid for an application to open the same object more than once. Each handle that is returned can be used for the functions for which the corresponding open was performed.
4. All name resolution within the local queue manager instance takes place at the time of the MQOPEN call. This may include one or more of the following for a given MQOPEN call:
   - Alias resolution to base queue name.
   - Resolution of remote queue name to remote queue manager name, and the local queue name by which it is known at the remote queue manager.

   However, be aware that subsequent MQINQ calls for the handle relate solely to the name that has been opened, and not to the object resulting after name resolution has occurred. For example, if the object opened is an alias, the attributes returned by the MQINQ call are the attributes of the alias, not the attributes of the base queue to which the alias resolves.

5. The attributes of an object can change while an application has the object open. Only InhibitPut and InhibitGet will be recognized. For all other attributes to be recognized, the queue must be closed and opened again.
6. A remote queue can be specified in one of two ways in the ObjDesc parameter of this call (see the ObjectName field in “MQOD - MQ object descriptor structure” on page 131):
   - By specifying ObjectName as the local resource-name of the remote queue, as known to the local queue manager. In this case, ObjectQMgrName refers to the connected queue manager. See “Queue name format” on page 30, for details.
   - By specifying ObjectName as the local resource-name of the remote queue, as known to the remote queue manager. In this case, ObjectQMgrName is the name of the remote queue manager.

   In either case:
   - No message flows occur at the time of an MQOPEN call to the remote queue manager to perform authorization checks.
7. [2]An MQOPEN call with the MQOO_BROWSE option establishes a browse cursor, for use with the MQGET calls that specify the object handle and one of the browse options. This allows the queue to be scanned without altering its contents. A message that has been found by browsing can subsequently be removed from the queue using the MQGMO_MSG_UNDER_CURSOR option.

   Each established browse cursor adversely impacts the performance of non-browse MQGET calls. It is recommended therefore that browse operations should be completed as rapidly as possible, and the cursor destroyed by closing the queue. If further browse operations are required later, it is better to close the queue and reopen it when needed, in order to establish a new browse cursor.

   Multiple browse cursors can be active for a single application issuing several MQOPEN requests for the same queue.

MQGET - get message

```
MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer, DataLength, CompCode,
Reason)
```

The MQGET call retrieves a message from a local queue that has been opened using an MQOPEN call.

For a queue that has been opened for getting, the application can get messages from that queue by means of the MQGET call.

The application specifies a partially filled-in message descriptor (MsgDesc), some options that control the action of the call (GetMsgOpts), an empty buffer (Buffer), and the length of the buffer (BufferLength).

The application receives in return the message data in the buffer (Buffer), and the total length of the message data (DataLength). The message descriptor (MsgDesc) is completed with information about the message just retrieved.

The MQGET call can be used repeatedly to get many messages from the same queue, without the intervening use of the MQOPEN and MQCLOSE calls.

Parameters

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

2. Scanning beyond the first message on the queue is not currently supported on UNIX.

Hobj (MQHOBJ) - input

Object handle.

This handle represents the queue from which a message is to be read. The queue must have been opened with one or more of the following options (see the MQOPEN call for details):

- MQOO_INPUT_SHARED
- MQOO_INPUT_EXCLUSIVE
- MQOO_BROWSE

MsgDesc (PMQMD) - input/output

Message descriptor.

This structure describes the attributes of the message required, and the attributes of the message retrieved. See MQMD in “MQMD - MQ message descriptor structure” on page 132, for the format of the message descriptor.

If BufferLength is less than the message length, MsgDesc is still filled in by the queue manager, whether or not MQGMO_ACCEPT_TRUNCATED_MSG is specified on the GetMsgOpts parameter (see the Options field in “MQGMO - MQGet message options structure” on page 138 for more information).

GetMsgOpts (PMQGMO) - input/output

Options that control the action of an MQGET call.

See MQGMO in “MQGMO - MQGet message options structure” on page 138 for details.

BufferLength (MQLONG) - input

Length in bytes of the Buffer area.

Buffer (MQBYTExBufferLength) - output

Area to contain the message data.

If BufferLength is less than the message length, as much of the message as possible is moved into Buffer, whether or not MQGMO_ACCEPT_TRUNCATED_MSG is specified on the GetMsgOpts parameter (see the Options field in “MQGMO - MQGet message options structure” on page 138 for more information). Unless MQGMO_ACCEPT_TRUNCATED_MSG is specified, the message is not deleted from the queue.

In the C programming language, the parameter is declared as a pointer-to-void; this means that the address of any type of data can be specified as the parameter.

If character data is used within the application message text, the coded character set identifier has to be agreed between the sending and receiving applications, or else the character set has to be limited to the subset that is known to occupy the same code points for both the sender and receiver.

If the buffer length parameter is zero, Buffer is not referenced; in this case, the parameter address passed by programs written in C can be null.

DataLength (PMQLONG) - output

Length of the message.

This is the length of the application data in the message. If this is greater than BufferLength, only BufferLength bytes are returned in the Buffer parameter (the message is truncated). If the value is zero, it means that the message contains no application data.

If BufferLength is less than the message length, DataLength is still filled in by the queue manager, whether or not MQGMO_ACCEPT_TRUNCATED_MSG is specified on the GetMsgOpts parameter (see the Options field in “MQGMO - MQGet message options structure” on page 138 for more information). This allows the application to determine the size of the buffer required to accommodate the message data.

CompCode (PMQLONG) - output

Completion code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_WARNING - Warning (partial completion).
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report.

If CompCode is MQCC_WARNING:

- MQRC_TRUNCATED_MSG_ACCEPTED - Truncated message returned (message deleted from queue).
- MQRC_TRUNCATED_MSG_FAILED - Truncated message returned (message not deleted from queue).
- MQRC_NO_MSG_LOCKED

If CompCode is MQCC_FAILED:

- MQRC_BUFFER_ERROR - Buffer parameter not valid.
- MQRC_BUFFER_LENGTH_ERROR - Buffer length parameter not valid.
- MQRC_CONNECTION_BROKEN - Connection lost.
- MQRC_CORREL_ID_ERROR - CorrelId field not set to MQCI_NONE.
- MQRC_DATA_LENGTH_ERROR - Data length parameter not valid.
- MQRC_GET_INHIBITED - Gets inhibited for the queue.
- MQRC_GMO_ERROR - Get Message error.
- MQRC_HCONN_ERROR - Connection handle not valid.
- MQRC_HOBJ_ERROR - Object handle not valid.
- MQRC_LOCK_NOT_AVAILABLE - No more locks available.
- MQRC_MD_ERROR - Message descriptor not valid.
- MQRC_MSG_ID_ERROR - MsgId field not set to MQMI_NONE.
- MQRC_NO_MSG_AVAILABLE - No message available.
- MQRC_NO_MSG_UNDER_CURSOR - Browse cursor not positioned on message.
- MQRC_NOT_OPEN_FOR_BROWSE - Queue object not open for browse.
- MQRC_NOT_OPEN_FOR_INPUT - Queue object not open for input.
- MQRC_OBJECT_CHANGED - Object definition changed since opened.
- MQRC_OPTIONS_ERROR - Options not valid or consistent.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_WAIT_INTERVAL_ERROR - Negative wait interval in MQGMO.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.

See “MQI return codes” on page 144, for more details.

Guidelines

1. The message retrieved is normally deleted from the queue as part of the MQGET call (Options set to 0). Messages can be read and not deleted by specifying the MQGMO_BROWSE_FIRST and MQGMO_LOCK options. This does though leave an outstanding record lock on the read record.
2. A locked record can be unlocked by again calling MQGET with the MQGMO_UNLOCK option set. The previously read message is now available for another process to read. Buffer and DataLength are not updated.
3. A locked record can be deleted by again calling MQGET with the MQGMO_MSG_UNDER_CURSOR option. This deletes and unlocks the previously read message.
4. Issuing another MQGET call while having an outstanding locked record assumes that the currently locked record is to be unlocked. If this is the only application reading this queue, then the same record will be returned.

MQPUT - put message

```
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer, CompCode, Reason)
```

The MQPUT call puts a message on a queue; the queue must already be open.

When the queue has been opened for putting, the application can put messages to that queue by means of the MQPUT call.

The application specifies information about the message to be put (MsgDesc), options that control the action of the put (PutMsgOpts), the length of the data (BufferLength), and the message itself (Buffer).

The MQPUT call can be used repeatedly to put many messages on the same queue, without intervening use of the MQOPEN and MQCLOSE calls.

Parameters

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

Hobj (MQHOBJ) - input

Object handle.

This handle represents the queue to which the message is added. The queue must be opened for MQOO_OUTPUT (see the MQOPEN call).

MsgDesc (PMQMD) - input/output

Message descriptor.

This structure describes the attributes of the message being sent, and receives feedback information after the put request is complete. See MQMD in “MQMD - MQ message descriptor structure” on page 132 for the format of the message descriptor.

PutMsgOpts (PMQPMO) - input/output

Options that control the action of the MQPUT call.

See MQPMO in “MQI data types and structures” on page 129 for details.

BufferLength (MQLONG) - input

Length of the message in Buffer.

Zero is valid, and indicates that the message contains no application data.

Buffer (MQBYTExBufferLength) - input

This is a buffer containing the application data to be sent.

In the C programming language, the parameter is declared as a pointer-to-void; this means that the address of any type of data can be specified as the parameter.

If character data is used within the application message text, the coded character set identifier has to be agreed between the sending and receiving applications, or else the character set used has to be limited to a subset that is known to occupy the same code points for both the sender and receiver.

If BufferLength parameter is zero, Buffer is not referenced; in this case, the parameter address passed by programs written in C can be null.

CompCode (PMQLONG) - output

Completion Code.

It is one of the following:

- `MQCC_OK` - Successful completion.
- `MQCC_WARNING` - Warning (partial completion).
- `MQCC_FAILED` - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is `MQCC_OK`:

- `MQRC_NONE` - No reason to report.

If CompCode is `MQCC_WARNING`:

- `MQRC_PRIORITY_EXCEEDS_MAXIMUM` - Priority exceeds maximum.

If CompCode is `MQCC_FAILED`:

- `MQRC_BUFFER_ERROR` - Buffer parameter not valid.
- `MQRC_BUFFER_LENGTH_ERROR` - Buffer length parameter not valid.
- `MQRC_CONNECTION_BROKEN` - Connection lost.
- `MQRC_EXPIRY_ERROR` - Expiry time not valid.
- `MQRC_FEEDBACK_ERROR` - Feedback code not valid.
- `MQRC_HCONN_ERROR` - Connection handle not valid.
- `MQRC_HOBJ_ERROR` - Object handle not valid.
- `MQRC_MD_ERROR` - Message descriptor not valid.
- `MQRC_MISSING_REPLY_TO_Q` - Missing reply-to queue.
- `MQRC_MSG_TOO_BIG_FOR_Q` - Message length greater than maximum for queue.
- `MQRC_MSG_TYPE_ERROR` - Message type in message descriptor not valid.
- `MQRC_PMO_ERROR` - Put-message-options structure not valid.
- `MQRC_NOT_OPEN_FOR_OUTPUT` - Queue object not open for output.
- `MQRC_OBJECT_CHANGED` - Object definition changed since opened.
- `MQRC_OPTIONS_ERROR` - Options not valid or not consistent.
- `MQRC_PERSISTENCE_ERROR` - Persistence not valid.
- `MQRC_PRIORITY_ERROR` - Priority not valid.
- `MQRC_PUT_INHIBITED` - Puts inhibited for queue.
- `MQRC_Q_FULL` - Queue already at maximum depth.
- `MQRC_Q_SPACE_NOT_AVAILABLE` - No space available on disk for queue.
- `MQRC_REPORT_OPTIONS_ERROR` - Report options in message descriptor not valid.

- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.

See “MQI return codes” on page 144, for more details.

Guidelines

1. The MQPUT call should be used when multiple messages are to be placed on a queue. An MQOPEN call, with the MQOO_OUTPUT attribute, is first issued, followed by one or more MQPUT requests to add messages to the queue. The queue is then closed with an MQCLOSE call.
2. If only one message is to be put on the queue, the MQPUT1 call can be used.

MQCLOSE - close object

```
MQCLOSE (Hconn, Hobj, Options, CompCode, Reason)
```

The MQCLOSE call relinquishes access to an object, and is the inverse of the MQOPEN call.

When the application has finished putting messages on a queue, or getting messages from a queue, the application must close the queue by means of the MQCLOSE call.

The application specifies the handle of the queue to be closed (Hobj), and some options that control the action of the call (Options). After the call, the queue handle (Hobj) is no longer valid, and messages cannot be put to the queue or removed from the application unless it performs another MQOPEN call.

An application that is reading from a queue does not have to empty the queue before closing it. Messages left on a queue are retained by the queue manager, and may be accessed later by the same or another application.

Parameters

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

Hobj(PMQHOBJ) - input/output

Object Handle.

This handle represents the object which is being closed. The value of Hobj was returned by a previous MQOPEN call.

On successful completion of the call, the queue manager sets this parameter to a value that is not a valid handle.

Options (MQLONG) - input

Options that control the action of an MQCLOSE call.

The following must be specified:

MQCO_NONE - No optional close processing required.

CompCode (PMQLONG) - input

Completion Code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report

If CompCode is MQCC_FAILED:

- MQRC_CONNECTION_BROKEN - Connection lost.
- MQRC_HCONN_ERROR - Connection handle not valid.
- MQRC_HOBJ_ERROR - Object handle not valid.
- MQRC_OPTIONS_ERROR - Options not valid or consistent.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.

See “MQI return codes” on page 144, for more details.

Guideline

When an application issues the MQDISC call, or ends either normally or abnormally, any objects which were opened by the application and which are still open are closed automatically with the MQCO_NONE option.

MQDISC - disconnect queue manager

```
MQDISC (Hconn, CompCode, Reason)
```

The MQDISC call breaks the connection between the queue manager and the application program, and is the inverse of MQCONN.

When the application has finished all interaction with the queue manager, the application must sever the connection by means of the MQDISC call.

After the call, the connection handle (Hconn) is no longer valid, and message-queuing calls cannot be issued by the application unless it performs another MQCONN call.

Parameters

Hconn (PMQHCONN) - input/output

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

On successful completion of the call, the queue manager sets this parameter to a value that is not a valid handle.

CompCode (PMQLONG) - output

Completion code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report.

If CompCode is MQCC_FAILED:

- MQRC_CONNECTION_BROKEN - Connection lost.
- MQRC_HCONN_ERROR - Connection handle not valid.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.

See “MQI return codes” on page 144, for more details.

Guideline

If an MQDISC call is issued when an application still has objects open, these objects are implicitly closed (with MQCO_NONE).

MQPUT1 - put one message

```
MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength, Buffer, CompCode, Reason)
```

The MQPUT1 call puts one message on a queue; the queue need not be open.

For some applications, the typical sequence of calls to MQOPEN, multiple MQPUTS, and finally MQCLOSE is an efficient method for putting many messages onto a queue. For applications where only a single put is required, such as a remote database update for a single record, the MQPUT1 call can be used.

The MQPUT1 call is equivalent in function to the sequence of an MQOPEN call, followed by an MQPUT, and finally an MQCLOSE call, but only requires a single call.

The application specifies the handle for the queue manager (Hconn), the queue to put the information (ObjDesc), information about the message to be put (MsgDesc), options that control the action of the put (PutMsgOpts), the length of the data (BufferLength), and the message itself (Buffer).

Parameters

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

ObjDesc (PMQOD) - input

Object descriptor.

This is a structure which identifies the queue to which the message is added. See MQOD in “MQOD - MQ object descriptor structure” on page 131 for the format of the object descriptor.

MsgDesc (PMQMD) - input/output

Message descriptor.

This structure describes the attributes of the message being sent, and receives feedback information after the put request is complete. See MQMD in “MQMD - MQ message descriptor structure” on page 132 for the format of the message descriptor.

PutMsgOpts (PMQPMO) - input/output

Options that control the action of the MQPUT1 call.

See MQPMO in “MQGMO - MQGet message options structure” on page 138 for details.

BufferLength (MQLONG) - input

Length of the message in Buffer.

Zero is valid, and indicates that the message contains no application data.

Buffer (MQBYTExBufferLength) - input

This is a buffer containing the application data to be sent.

In the C programming language, the parameter is declared as a pointer-to-void; this means that the address of any type of data can be specified as the parameter.

If character data is used within the application message text, the coded character set identifier has to be agreed between the sending and receiving applications, or else the character set used has to be limited to a subset that is known to occupy the same code points for both the sender and receiver.

If BufferLength parameter is zero, Buffer is not referenced; in this case, the parameter address passed by programs written in C can be null.

CompCode (PMQLONG) - output

Completion Code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_WARNING - Warning (partial completion).
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report.

If CompCode is MQCC_WARNING:

- MQRC_EXCEEDS_MAXIMUM - Priority exceeds maximum.

If CompCode is MQCC_FAILED:

- MQRC_ALIAS_BASE_Q_TYPE_ERROR - Alias base queue not a valid type.
- MQRC_BUFFER_ERROR - Buffer parameter not valid.
- MQRC_BUFFER_LENGTH_ERROR - Buffer length parameter not valid.
- MQRC_CONNECTION_BROKEN - Connection lost.
- MQRC_EXPIRY_ERROR - Expiry time not valid.
- MQRC_FEEDBACK_ERROR - Feedback code not valid.
- MQRC_HANDLE_NOT_AVAILABLE - No more handles available.
- MQRC_HCONN_ERROR - Connection handle not valid.
- MQRC_MD_ERROR - Message descriptor not valid.
- MQRC_MISSING_REPLY_TO_Q - Missing reply-to queue.
- MQRC_MSG_TOO_BIG_FOR_Q - Message length greater than maximum for queue.
- MQRC_MSG_TYPE_ERROR - Message type in message descriptor not valid.
- MQRC_NOT_AUTHORIZED - Not authorized for access.
- MQRC_OBJECT_CHANGED - Object definition changed since opened.
- MQRC_OBJECT_TYPE_ERROR - Object type not valid.
- MQRC_OD_ERROR - Object descriptor structure not valid.

- MQRC_OPTIONS_ERROR - Options not valid or not consistent.
- MQRC_PERSISTENCE_ERROR - Persistence not valid.
- MQRC_PMO_ERROR - Put-message-options structure not valid
- MQRC_PRIORITY_ERROR - Priority not valid.
- MQRC_PUT_INHIBITED - Puts inhibited for queue.
- MQRC_Q_FULL - Queue already at maximum depth.
- MQRC_Q_SPACE_NOT_AVAILABLE - No space available on disk for queue.
- MQRC_REPORT_OPTIONS_ERROR - Report options in message descriptor not valid.
- MQRC_STORAGE_NOT_AVAILABLE - Insufficient storage available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred
- MQRC_UNKNOWN_ALIAS_BASE_Q - Unknown alias base queue.
- MQRC_UNKNOWN_OBJECT_NAME - Unknown object name.
- MQRC_UNKNOWN_OBJECT_Q_MGR - Unknown object queue manager.
- MQRC_UNKNOWN_REMOTE_Q_MGR - Unknown remote queue manager.
- MQRC_XMIT_Q_TYPE_ERROR - Transmission queue not local.
- MQRC_XMIT_Q_USAGE_ERROR - Transmission queue with wrong usage.

See “MQI return codes” on page 144, for more details.

Guidelines

1. The MQPUT1 call can be used when a single message is to be added to a queue. It is functionally equivalent to the MQOPEN, MQPUT, MQCLOSE sequence of calls.
2. If several messages are to be added to the same queue, it is advisable to open the queue explicitly using an MQOPEN, call and then use repeated MQPUT calls before closing the queue using an MQCLOSE call, because this gives better performance than repeated use of the MQPUT1 call.

MQINQ - inquire about object attributes

```
MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
CharAttrLength, CharAttrs, CompCode, Reason)
```

The MQINQ call returns an array of integers and a set of character strings that contain the attributes of a specified queue.

Sometimes an application needs to determine one or more of the properties of a queue, in order to take appropriate action. For example, a load-balancing program might want to determine the current depth of the queue (that is the number of messages on the queue), so that the application could start another server if the number of queued messages has exceeded the capacity of the current number of servers.

The attributes of the queue can be determined by means of the MQINQ call.

The application specifies the queue whose attributes are to be queried (Hobj), the number of attributes required (SelectorCount), and the selector codes for those attributes (Selectors). The application receives in return the values for those attributes (IntAttrs and CharAttrs).

In order to use the MQINQ call, the queue must first be opened for inquiry using the MQOPEN call.

Parameters

Hconn (MQHCONN) - input

Connection handle.

This handle represents the connection to the queue manager, and is returned by the MQCONN call.

Hobj (MQHOBJ) - input

Object handle.

This handle represents the object whose attributes are required. The handle must have been returned by an MQOPEN call with the MQOO_INQUIRE option.

SelectorCount (MQLONG) - input

Count of selectors.

This is the count of selectors that are supplied in the Selectors array. It is the number of attributes that are to be returned. Zero is a valid value. The maximum value allowed is 256.

Selectors (MQLONGxSelectorCount) - input

Array of attribute selectors.

This is an array of SelectorCount attribute selectors; each selector identifies an attribute (integer or character) whose value is required.

Each selector must be valid for the type of object that Hobj represents. If the object is a queue, and the selector is:

- Not a valid selector for queues of any type, an error is raised.
- Only applicable to queues of type, or types, other than that of the object, the call completes with a warning.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (MQIA_* selectors) are returned in IntAttrs in the same order in which these selectors (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Attribute values that correspond to character attribute selectors (MQCA_* selectors) are returned in CharAttrs in the same order in which those selectors occur. MQIA_* selectors can be interleaved with the MQCA_* selectors; only the relative order within each type is important.

If all the MQIA_* selectors occur first, the same element numbers can be used to address corresponding elements in the Selectors and IntAttrs arrays.

For each MQCA_* selector in the following descriptions, the constant that defines the length in bytes of the resulting string CharAttrs is given.

The following are valid for *any queue type:*

- MQIA_Q_TYPE - This attribute requests the type of this queue. The value returned can be one of the following: MQQT_LOCAL, MQQT_ALIAS, and MQQT_REMOTE. This attribute is valid for all queue types and requires a field width of one longword.
- MQIA_INHIBIT_PUT - This attribute requests the setting of the put inhibit flag for this queue. A value of MQQA_PUT_INHIBITED is returned if puts are inhibited. Otherwise MQQA_PUT_ALLOWED is returned. This attribute is valid for all queue types and requires a field width of one longword.
- MQIA_DEF_PERSISTENCE - This attribute requests the default persistence of this queue. For this implementation MQPER_PERSISTENT is always returned. This attribute is valid for all queue types and requires a field width of one longword.
- MQCA_Q_NAME - This attribute requests the name of the currently opened queue. This attribute is valid for all queue types and requires a field width of 48 characters.
- MQCA_Q_DESC - This attribute requests the queue description. The data for this field comes from the Description field of the Create Queue Screen. This attribute is valid for all queue types and requires a field width of 64 characters.

The following are valid for *local queues*:

- MQIA_MAX_Q_DEPTH - This attribute requests the maximum queue depth of this queue. The value for this attribute comes from the value assigned to the Max Queue Depth field of the Create Local Queue Screen of MQM. A value of 999999999 means that this queue has unlimited depth. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_MAX_MSG_LENGTH - This attribute requests the maximum message length of this queue. The value for this attribute comes from the value assigned to the Max Message field of the Create Local Queue Screen of MQM. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_SHAREABILITY - This attribute requests the shareability of this queue. For this implementation all queues are shareable so MQQA_SHAREABLE is always returned. This attribute is valid for local queues and requires a field width of one longword.
- MQIA_DEFINITION_TYPE - This attribute requests the definition type of this queue. For this implementation MQQDT_PREDEFINED is always returned. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_USAGE - This attribute requests the usage of this queue. The value returned is one of MQUS_NORMAL, meaning this is a local queue used for application purposes, or MQUS_TRANSMISSION, meaning this is a local queue used only for transmission purposes. The difference being that applications using MQI cannot open a local queue with usage = MQUS_TRANSMISSION for input of any kind (MQOO_INPUT_SHARED, MQOO_INPUT_EXCLUSIVE, or MQOO_BROWSE). This attribute is only valid for local queues and requires a field width of one longword.
- MQIA_TRIGGER_CONTROL - This attribute requests whether trigger control is on or off for this queue. Since triggering is not supported for this platform, the value MQTC_OFF is always returned. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_TRIGGER_TYPE - This attribute requests the trigger type available. Since triggering is not supported for this platform, the value MQTT_NONE is always returned. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_OPEN_INPUT_COUNT - This attribute requests the total number of times this queue has been opened for input and not subsequently closed. As a note, the sum of the values returned by MQIA_OPEN_INPUT_COUNT plus MQIA_OPEN_OUTPUT_COUNT does not equal the total number of queue users. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_OPEN_OUTPUT_COUNT - This attribute requests the total number times this queue has been opened for output and not subsequently closed. As a note, the sum of the values returned by MQIA_OPEN_INPUT_COUNT plus MQIA_OPEN_OUTPUT_COUNT does not equal the total number of queue users. This attribute is valid only for local queues and requires a field width of one longword.
- MQIA_CURRENT_Q_DEPTH - This attribute requests the current queue depth. This value is not to be confused with the total number of records on the queue. This value represents the number of non-deleted messages on the queue. This number will not exceed the maximum queue depth assigned with this queue. This attribute is valid for local queues and requires a field width of one longword.
- MQCA_PROCESS_NAME - Though supported by the MQI interface, this attribute request will only return a space-filled buffer as UNIX does not support process names. This attribute is valid for all queue types and requires a field width of 64 characters.
- MQCA_INITIATION_Q_NAME - This attribute, though supported by the MQI interface, is only filled with spaces in this implementation. This attribute is only valid for local queues and requires a field width of 48 characters.
- MQCA_CREATION_DATE - This attribute requests the creation date of a queue. This is the date on which the physical queue was created through MQM. Since physical queues exist for local queues only, this attribute is only valid for local queues and requires a field width of 12 characters.

- MQCA_CREATION_TIME - This attribute requests the creation time of a queue. This is the time at which the physical queue was created through MQM. Since physical queues exist for local queues only, this attribute is only valid for local queues and requires a field width of 8 characters.
- MQIA_INHIBIT_GET - This attribute requests the setting of the get inhibit flag for this queue. A value of MQQA_GET_INHIBITED is returned if gets are inhibited. Otherwise MQQA_GET_ALLOWED is returned. The attribute is valid for all queue types and requires a field width of one longword.

The following are valid for *remote queues:*

- MQCA_REMOTE_Q_MGR_NAME - This attribute requests the name of the Remote Queue Manager. The data for this attribute comes from the Remote Queue Manager field of the Create Remote Queue Screen. This attribute is valid only for remote queues and requires a field width of 48 characters.
- MQCA_REMOTE_Q_NAME - This attribute requests the name of the Remote Queue. The data for this attribute comes from the Remote Queue field of the Create Remote Queue Screen. This attribute is valid only for remote queues and requires a field width of 48 characters.
- MQCA_XMIT_Q_NAME - This attribute requests the name of the transmit queue associated with either a remote queue definition or a Queue Manager alias definition. The data for this field comes from either the Transmit Queue field of the Create Remote Queue Screen or the Create Queue Manager Alias Screen (depending on the queue type). This attribute is valid only for remote queues and queue manger aliases. It requires a field width of 48 characters.

The following are valid for *alias queues*:

- MQCA_BASE_Q_NAME - This attribute requests the base queue name of an alias queue. When an alias queue was created using MQM, this was the value of the field labeled Alias To. This attribute is only valid for alias queues and requires a field width of 48 characters.
- MQIA_INHIBIT_GET - This attribute requests the setting of the get inhibit flag for this queue. A value of MQQA_GET_INHIBITED is returned if gets are inhibited. Otherwise MQQA_GET_ALLOWED is returned. The attribute is valid for all queue types and requires a field width of one longword.

IntAttrCount (MQLONG) - input

Count of integer attributes.

This is the number of elements in the IntAttrs array. Zero is a valid value if there are no MQIA_* selectors in Selectors.

If this is at least the number of MQIA_* selectors in the Selectors parameter, all integer attributes requested are returned.

IntAttrs (MQLONGxIntAttrCount) - output

This is an array of IntAttrCount integer attribute values.

Integer attribute values are returned in the same order as the MQIA_* selectors in the Selectors parameter. If the array contains more elements than the number of MQIA_* selectors, the excess elements are unchanged.

If Hobj represents a queue, but an attribute selector is not applicable to that type of queue, the specific value MQIAV_NOT_APPLICABLE is returned for the corresponding element in the IntAttrs array.

If the IntAttrCount or SelectorCount parameter is zero, IntAttrs is not referenced; in this case, the parameter address passed by programs written in C may be null.

CharAttrLength (MQLONG) - input

Length of character-attributes buffer.

This is the length in bytes of the CharsAttrs parameter.

This must be at least the sum of the lengths required to hold each attribute string (see Selectors). Zero is a valid value if there are no MQCA_* selectors in Selectors.

CharAttrs (MQCHARxCharAttrLength) - output

Character attributes.

This is the buffer in which the character attributes are returned, concatenated together. The length of the buffer is given by the CharAttrLength parameter.

Character attributes are returned in the same order as the MQCA_* selectors in the Selectors parameter. The length of each attribute string is fixed for each attribute (see Selectors), and the value in it is padded to the right with blanks if necessary.

If the buffer is larger than is needed to contain all of the requested character attributes (including padding), the excess, beyond the last attribute returned, is unchanged.

If Hobj represents a queue, but an attribute selector is not applicable to that type of queue, a character string consisting entirely of asterisks (*) is returned as the value of that attribute in CharAttr.

If the CharAttrLength or SelectorCount parameter is zero, CharAttrs is not referenced; in this case, the parameter address passed by programs written in C may be null.

CompCode (PMQLONG) - output

Completion code.

It is one of the following:

- MQCC_OK - Successful completion.
- MQCC_WARNING - Warning (partial completion).
- MQCC_FAILED - Call failed.

Reason (PMQLONG) - output

Reason code qualifying CompCode.

If CompCode is MQCC_OK:

- MQRC_NONE - No reason to report.

If CompCode is MQCC_WARNING:

- MQRC_INT_ATTR_COUNT_TOO_SMALL - Not enough space allowed for integer attributes.
- MQRC_CHAR_ATTRS_TOO_SHORT - Not enough space allowed for character attributes.
- MQRC_SELECTOR_NOT_FOR_TYPE - Selector not applicable to queue type.

If CompCode is MQCC_FAILED:

- MQRC_CHAR_ATTR_LENGTH_ERROR - Length of character attributes not valid.
- MQRC_CHAR_ATTRS_ERROR - Character Attribute string not valid.
- MQRC_CONNECTION_BROKEN - Connection lost.
- MQRC_HCONN_ERROR - Connection handle not valid.
- MQRC_HOBJ_ERROR - Object handle not valid.
- MQRC_INT_ATTR_COUNT_ERROR - Count of integer attributes not valid.
- MQRC_INT_ATTRS_ARRAY_ERROR - Integer attributes array not valid.
- MQRC_NOT_OPEN_FOR_INQUIRE - Queue object not open for inquire.
- MQRC_OBJECT_CHANGED - Object definition changed since opened.
- MQRC_SELECTOR_COUNT_ERROR - Count of selectors not valid.
- MQRC_SELECTOR_ERROR - Attribute selector not valid.
- MQRC_SELECTOR_LIMIT_EXCEEDED - Count of selectors too big.
- MRQC_STORAGE_NOT_AVAILABLE - Storage not available.
- MQRC_UNEXPECTED_ERROR - Unexpected error occurred.

See “MQI return codes” on page 144, for more details.

Guidelines

1. The values returned are a snapshot of the selected attributes. There is no guarantee that the attributes will not change before the application can act upon the returned values.
2. See “MQSeries System configuration elements” on page 29, for more information about queue types and attributes.
3. If more than one of the warning situations arise (see the CompCode parameter), the first one of the following reasons that applies is returned:
   - MQRC_SELECTOR_NOT_FOR_TYPE
   - MQRC_INT_ATTR_COUNT_TOO_SMALL
   - MQRC_CHAR_ATTRS_TOO_SHORT

MQI data types and structures

This section will examine the data types used by the MQI and will then present the primary data structures important to the MQI functions.

Data types

The following data types are used by the message queuing services in the MQSeries System:

- Elementary
- Structure

These data types correspond to data types that could be declared in a language that supports user-defined data types, such as the C programming language.

All user-defined data types ultimately resolve to elementary data types, or to aggregates of elementary types (arrays or structures).

Elementary data types

Message queuing uses the following elementary data types:[3]

- MQBYTE - A single byte (string of eight bits)
- MQCHAR - A single character in a defined character set
- MQLONG - A four-byte signed binary integer

3. Some function parameters in C are defined as being pointers to the appropriate data type.

MQBYTE - Byte

The MQBYTE data type represents a single byte of data. No particular interpretation is placed on the byte. The byte is treated as a string of bits, and not as a character or binary number. No special alignment is required.

An array of MQBYTE is sometimes used to represent an area of main storage whose nature is not known to the queue manager. For example, the area may contain application message data or a structure. The boundary alignment of this area must be compatible with the nature of the data it contains.

In the C programming language, any data type can be used for function parameters that are shown as arrays of MQBYTE. Such parameters are always passed by address, and in C the function parameter is declared as a pointer-to-void.

MQBYTE24 - String of 24 Bytes

A string of 24 bytes. Each byte is described by the MQBYTE data type.

MQCHAR - Character

The MQCHAR data type represents a single character. The coded character set identifier of the character is that of the queue manager. No special alignment is required.

Note: Application message data specified on MQGET, MQPUT, and MQPUT1 calls is described by the MQBYTE data type.

MQCHARn - String of n Characters

Each MQCHARn data type represents a string of *n* characters, where *n* can take one of the following values:

4, 8, 12, 16, 28, 32, 48, 64, 128, 256

Each character is described by the MQCHAR data type. No special alignment is required.

If the data in the string is shorter than the defined length of the string, the data must be padded with blanks to fill the string. In some cases, a null character can be used to end the string prematurely, instead of padding with blanks.

Characters beyond the null character, up to the defined length of the string, are ignored. Cases where null characters may be used are identified in the call and data type descriptions.

When the queue manager returns character strings to the application (for example, on the MQGET call), the queue manager always pads with blanks to the defined length of the string.

Constants are available that define the lengths of the character string fields.

MQHCONN - Connection Handle

The MQHCONN data type represents a *queue manager* connection handle. The MQHCONN data type is defined as an MQLONG, and must be aligned on a 4-byte boundary.

Applications must only test variables of this type for equality.

MQHOBJ - Object Handle

The MQHOBJ data type represents an object (*queue*) handle. The MQHOBJ is defined as an MQLONG, and must be aligned on a 4-byte boundary.

Applications must only test variables of this type for equality.

MQLONG - Long Integer

The MQLONG data type is a 32-bit signed binary integer that can take any value in the range -2147483648 through +2147483647, unless otherwise restricted by the context. This data type is sometimes referred to as a "longword".

Structure data types

The supported programming languages vary in their functionality with respect to structures, and certain rules and conventions are adopted in mapping the message-queuing structure data types of each programming language.

Boundary alignments

1. Structures are aligned on their natural boundaries. All message-queuing structures require 4-byte alignment.
2. Each field in the structure is aligned on its natural boundary. Fields of type MQLONG are aligned on 4-byte boundaries. Other fields are aligned on 1-byte boundaries.
3. The length of a structure is a multiple of its boundary requirement. All message-queuing structures have lengths that are multiples of four bytes.
4. Padding fields are declared explicitly where necessary to ensure compliance with rules 2 and 3.

References to structure components

The supported programming languages allow references to structure components to be qualified with the name of the structure. Multiple instances of the structure may be declared:

- C has dot-qualification

Characters in names

The supported programming languages accept mixed case, however, the following points should be noted:

- The C language is case-sensitive, and so the names of data types, structures fields, and named constants must be coded precisely as shown in this guide.

MQOD - MQ object descriptor structure

The MQOD structure is used to specify a queue object.

This structure is passed as a parameter to the MQOPEN and MQPUT1 calls.

StrucId (MQCHAR4)

Structure identifier.

The value must be:

MQOD_STRUC_ID

Structure identifier for Object Descriptor.

This is always an input field.

Version (MQLONG)

Structure version number.

The value must be:

MQOD_VERSION_1

Structure version number for Object Descriptor.

This is always an input field.

ObjectType (MQLONG)

Object type.

Type of object being named in ObjectName. This must be:

MQOT_Q

Queue.

This is always an input field.

ObjectName (MQCHAR48)

Object name.

The local name of the object as defined on the queue manager identified by ObjectQMgrName.

The name must not contain leading or embedded blanks, but may contain trailing blanks. The first null character and characters following it are treated as blanks.

This is an input field.

ObjectQMgrName (MQCHAR48)

Object queue manager name.

The name of the queue manager on which the ObjectName object is defined.

If the name is specified, it must not contain leading or embedded blanks, but may contain trailing blanks. The first null character and characters following it are treated as blanks.

A name which is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected.

This is an input field.

DynamicQName (MQCHAR48)

This is a reserved field.

AlternateUserId (MQCHAR12)

This is a reserved field.

MQMD - MQ message descriptor structure

The MQMD structure is used to describe the attributes of a message. It is an input/output variable for MQGET, MQPUT, and MQPUT1 calls.

StrucId (MQCHAR4)

Structure identifier.

The value must be:

MQMD_STRUC_ID

Structure identifier for Message Descriptor.

This is always an input field.

Version (MQLONG)

Structure version number.

The value must be:

MQMD_VERSION_1

Structure version number for Message Descriptor.

This is always an input field.

Report (MQLONG)

Reserved.

This is a reserved field. The value must be 0 (zero).

MsgType (MQLONG)

Message type.

This indicates the type of the message. It must be one of the following:

- MQMT_REQUEST - Message requiring reply (requires non-blank ReplyToQ).
- MQMT_REPLY - A reply to earlier request message.
- MQMT_DATAGRAM - A message not requiring a reply.
- MQMT_REPORT - A report message.

The description for these options follow.

MQMT_REQUEST

This message is one requiring a reply.

MQMT_REPLY

This message is the reply to an earlier request message (MQMT_REQUEST). The message should be sent to the queue indicated by the ReplyToQ field of the request message.

Note: The queue manager does not enforce the request-reply relationship. The request-reply relationship is the responsibility of the application.

MQMT_DATAGRAM

The message is one which does not require a reply.

MQMT_REPORT

The message is reporting on some unexpected occurrence (for example, a request message was received which contained data which was not valid).

The message should be sent to the queue indicated by ReplyToQ field of the message descriptor of the message which caused the error.

The Feedback field should be set to indicate the nature of the report. In addition, the CorrelId field of the report message should be set to the message identifier of the message which caused the error.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls.

Expiry (MQLONG)

Reserved.

This is a reserved field. The value must be -1.

Feedback (MQLONG)

Feedback code.

This is used with a message of type MQMT_REPORT to indicate the nature of the report, and is only meaningful with that type of message.

Feedback codes are grouped as follows:

- MQFB_NONE - No feedback provided.
- MQFB_SYSTEM_FIRST - Lowest value for system-generated feedback.
- MQFB_SYSTEM_LAST - Highest value for system-generated feedback.

Note: No feedback codes are generated by the queue manager.

- MQFB_APPL_FIRST - Lowest value for application-generated feedback.
- MQFB_APPL_LAST - Highest value for application-generated feedback.

Applications which generate report messages should not use feedback codes in the system range, other than MQFB_QUIT.

On MQPUT or MQPUT1 calls, the value specified must be within either the system range or the user range.

A special feedback code is:

MQFB_QUIT

Application should end. This can be used by a workload scheduling program to control the number of instances of an application program that are running. Sending an MQMT_REPORT message with this feedback code to an instance of the application program indicates to that instance that it should stop processing. However, adherence to this convention is a matter for the application. It is not enforced by the queue manager.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls.

Encoding (MQLONG)

Data encoding.

This identifies the representation used for the numeric values in the application message data. This applies to binary integer data, packed-decimal integer data, and floating-point data.

The following value is defined:

MQENC_NATIVE

Native machine encoding.

The encoding is the same as that of the machine on which the application is running.

Note: The value of this constant is environment-specific.

Applications should normally specify the MQENC_NATIVE.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls.

CodedCharSetId (MQLONG)

Coded character-set identifier.

This specifies the coded character-set identifier of character data in the user message data.

Note that character data in the message descriptor and the other message queuing data structures must be in the character set used by the queue manager.

The following special value may be specified:

MQCCSI_Q_MGR

Queue manager's coded character-set identifier.

Character data in the user message data is in the queue manager's character set.

On MQPUT and MQPUT1 calls, the queue manager changes the value MQCCSI_Q_MGR to the value of the queue manager's CodedCharSetId attribute. MQCCSI_Q_MGR is never returned by the MQGET call.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls.

Format (MQCHAR8)

Format name.

This is the name that the sender of the message may use to indicate to the receiver the nature of the data in the message. Any characters that are in the queue manager's character set may be specified for the name, but it is recommended that the name be restricted to the following:

- Uppercase A through Z
- Numeric digits 1 through 9

If other characters are used, it may not be possible to translate the name between the character sets of the sending and receiving queue managers.

Note: Do not use names beginning with "MQ". Names beginning with "MQ" are reserved for use by the queue manager.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls.

Priority (MQLONG)

Reserved.

This is a reserved field. The value for this field must be 0 (zero). Values greater than zero are accepted with a warning.

Persistence (MQLONG)

Message persistence.

For MQPUT and MQPUT1 calls, the value must be one of the following:

- MQPER_PERSISTENT - Message is persistent. The message survives restarts of the queue manager. When a persistent message is sent to a remote queue, a store-and-forward mechanism is used to hold the message on a local queue manager instance until it is known to have arrived at the next destination.
- MQPER_NOT_PERSISTENT [4]- Message not persistent. The message does not survive restarts of the queue manager. Note that an MQPER_NOT_PERSISTENT message found on a direct access storage device (DASD) is discarded on a restart, even if an intact copy of the message is found on the DASD during restart.
- MQPER_PERSISTENCE_AS_Q_DEF - Message has default persistence. The persistence for the message is taken from the DefPersistence attribute for the target queue. If the target is an alias queue, the default persistence for that target queue is used, not that for the basic queue to which the message is actually delivered.

For an MQGET call, the value returned is either MQPER_PERSISTENT or MQPER_NOT_PERSISTENT.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls.

MsgId (MQBYTE24)

Message identifier.

On return from an MQGET call, the MsgId field is set to the message identifier of the message returned (if any).

For MQPUT and MQPUT1 calls, if MQMI_NONE is specified by the application, the queue manager generates a unique message identifier[5] that it places in the message descriptor sent with the message.

The queue manager also returns this message identifier in the message descriptor belonging to the sending application. The application can use this value to record information about particular messages, and to respond to queries from other parts of the application.

The sending application can also specify a particular value for the message identifier, other than MQMI_NONE. This stops the queue manager generating a unique message identifier. This facility can be used by an application that is forwarding a message, to propagate the message identifier of the original message.

The queue manager does not itself make any use of this field except to:

- Generate a unique value if requested.
- Deliver the value to the application that issued the get request for the message.

This field is not subject to any translation based on the character set of the queue manager. The field is treated as a string of bits.

The following special value may be used:

MQMI_NONE

No message identifier is specified. The value is binary zero for the length of the field.

For the MQGET call, MQMI_NONE must be specified, and the first available message on the queue will be returned.

---

4. MQPER_NOT_PERSISTENT is not supported. MQPER_NOT_PERSISTENT must not be specified by an application putting a message. However, this value (zero) may be returned after an MQGET call, if the message was originally put at a queue manager which does support this option (for example, MQSeries for MVS/ESA). The message, nevertheless, behaves as if it is persistent on the MQSeries System.
5. A generated MsgId consists of a 4-byte product identifier followed by a product-specific implementation of a unique number. There is no guarantee that queue manger-generated MsgId values do not clash with application-generated ones.

This is an input-output field for MQGET, MQPUT and MQPUT1 calls.

CorrelId (MQBYTE24)

Correlation identifier.

For MQPUT and MQPUT1 calls, the application can specify any value. The queue manager transmits this value with the message and delivers it to the application that issued the get request for the message.

The field is not subject to any translation based on the character set of the queue manager. The field is treated as a string of bits.

The following special value may be used:

MQCI_NONE

No correlation identifier is specified.

The value is binary zero for the length of the field.

For the MQGET call, MQCI_NONE must be specified, and the first available message on the queue will be returned.

This is an input-output field for MQGET calls, and an input field for MQPUT and MQPUT1 calls.

BackoutCount (MQLONG)

This is a reserved field.

ReplyToQ (MQCHAR48)

Name of reply queue.

The name of the message queue to which the application that issued the get request for the message should send MQMT_REPLY and MQMT_REPORT messages. The name is the local name of a queue that is defined on the queue manager identified by ReplyToQMgr.

For MQPUT and MQPUT1 calls, this field is required if an MQMT_REQUEST type message is specified in the message descriptor. However, the value specified is passed on to the application that issued the get request for the message, whatever the message type.

If the name is specified, it should not contain leading or embedded blanks, but it may contain trailing blanks. The first null character and characters following the null are treated as blanks. A name that is entirely blank up to the first null character or the end of the field indicates that there is no reply-to-queue.

For the MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

The queue specified must be able to be opened for output by the application that receives the request message. The application design must ensure that the necessary queues exist and are appropriately authorized.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls.

For MQPUT and MQPUT1, the name is checked to be non-blank if the message type is MQMT_REQUEST. If the name is non-blank, and ReplyToQMgr is blank, the queue manager attempts to resolve the specified ReplyToQ. If the name will resolve, it will be substituted, and may be seen by the application issuing the put. If the specified name cannot be resolved, it is kept as is.

ReplyToQMgr (MQCHAR48)

Name of the reply queue manager.

The name of the queue manager to which the reply message is sent. ReplyToQ is the local name of a queue that is defined to that queue manger.

If the name is specified, it should not contain leading or embedded blanks, but it may contain trailing blanks. The first null character and characters following it are treated as blanks. A name that is entirely blank up to the first null character or the end of the field denotes the queue manager to which the application is connected.

For an MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls.

For MQPUT and MQPUT1 requests, if ReplyToQMgr is blank and ReplyToQ is non-blank, the queue manager attempts to resolve the specified ReplyToQ. If the name will resolve, ReplyToQMgr is replaced with the resolved queue-manager name. If it will not resolve, it is set to the name of the connected queue manager.

UserIdentifier (MQCHAR12)

This is a reserved field.

AccountingToken (MQBYTE32)

This is a reserved field.

ApplIdentityData (MQCHAR32)

This is a reserved field.

PutApplType (MQLONG)

This is a reserved field.

PutApplName (MQCHAR28)

This is a reserved field.

PutDate (MQCHAR8)

This is a reserved field.

PutTime (MQCHAR8)

This is a reserved field.

ApplOriginData (MQCHAR4)

This is a reserved field.

MQPMO - MQPut message options structure

StrucId (MQCHAR4)

Structure identifier.

The value must be:

MQPMO_STRUC_ID

Structure identifier for Put-Message Options.

This is always an input field.

Version (MQLONG)

Structure version number.

The value must be:

MQPMO_VERSION_1

Structure version number for Put-Message Options.

This is always an input field.

Options (MQLONG)

This is a reserved field. This field must always be 0 (zero) or MQPMO_NO_SYNCPOINT.

Timeout (MQLONG)

This is a reserved field.

Context (MQHOBJ)

This is a reserved field.

KnownDestCount (MQLONG)

This is a reserved field.

UnknownDestCount (MQLONG)

This is a reserved field.

InvalidDestCount (MQLONG)

This is a reserved field.

ResolvedQName (MQCHAR48)

Resolved name of the destination queue.

This is an output field that is set by the queue manager to the name of the queue that received the message after alias resolution. This can be either a local queue name or a remote queue name.

In each case the name is the local name of a queue that is defined on the queue manager identified by ResolvedQMgrName.

ResolvedQMgrName (MQCHAR48)

Resolved name of destination queue manager.

The name of the queue manager that received the message after alias resolution. ResolvedQName is the local name of a queue that is defined on that queue manager.

This is an output field.

MQGMO - MQGet message options structure

The MQGMO structure is an input variable for passing the MQGET call.

StrucId (MQCHAR4)

Structure identifier.

The value must be:

MQGMO_STRUC_ID

Structure identifier for Get-Message Options.

This is always an input field.

Version (MQLONG)

Structure version number.

The value must be:

MQGMO_VERSION_1

Structure version number for Get-Message Options.

This is always an input field.

Options (MQLONG)

Options.

Any or none of the following can be specified. If more than one is required, the values are added together.[6] Combinations that are not valid are noted. All other combinations are valid. The following options are supported:

- MQGMO_NONE - No options.
- MQGMO_WAIT - Wait for message to arrive.

6. Do not add the same constant more than once.

- MQGMO_NO_WAIT - Return immediately if no suitable message.
- MQGMO_NO_SYNCPOINT - No effect, as syncpoint is not supported.
- MQGMO_LOCK - Message get with lock (only valid with MQGMO_BROWSE_FIRST).
- MQGMO_BROWSE_FIRST - Browse from start of queue (Only valid with MQQGMO_LOCK).
- MQGMO_UNLOCK - Unlock message previously locked with MQGMO_BROWSE_FIRST and MQGMO_LOCK.
- MQGMO_ACCEPT_TRUNCATED_MSG - Allow truncation of message data.
- MQGMO_MSG_UNDER_CURSOR - Delete message previously locked with MQGMO_BROWSE_FIRST and MQGMO_LOCK.

The description of these options follows.

MQGMO_NONE

The MQGET call reads the next message from the queue.

If get requests are inhibited, this call returns with an error.

MQGMO_WAIT

The application is to wait until a message arrives. The maximum time the application waits is specified in WaitInterval.

If get requests are inhibited, this call returns with an error, whether or not there are any messages on the queue. If get requests become inhibited while this call is waiting, it returns immediately with an error.

This option can be used with the MQGMO_BROWSE_FIRST option.

If several applications are waiting on the same shared queue, one application will be activated when a suitable message arrives.

The following points should be noted:

- It cannot be predicted which application is selected. In particular, the application waiting longest is not necessarily the one selected.
- Path length and operating system priority-scheduling considerations can mean that a waiting application of lower priority than expected retrieves the message.
- It may also happen that an application that is not waiting retrieves the message in preference to one that is waiting.

Notes:

1. Specific get-wait requests are not supported.
2. It is an error to specify MQGMO_SET_SIGNAL[7] with the MQGMO_WAIT option. It is also an error to specify this option with a queue handle for which a signal is outstanding.

MQGMO_NO_WAIT

The application is not to wait if no suitable message is available. This is the opposite of the MQGMO_WAIT option, and is defined to aid program documentation. It is the default if neither is specified.

MQGMO_NO_SYNCPOINT

This option exists for compatibility. Syncpoint is not supported in this implementation.

MQGMO_LOCK

This option is only valid in conjunction with MQGMO_BROWSE_FIRST. MQOO_BROWSE must have been included in the open options when the handle was opened.

MQGMO_BROWSE_FIRST

The first message in the queue is returned to the application, but remains in the queue. A lock remains outstanding on this record until either MQGET is called again with MQGMO_MSG_UNDER_CURSOR (which will delete the record), MQGMO_UNLOCK (which will unlock this message thus making it available for another application), or again with

7. MQGMO_SET_SIGNAL is not supported in this release.

MQGMO_BROWSE_FIRST + MQGMO_LOCK (this will unlock the message). Should this application be the only one reading this queue, this will again read the same message establishing a lock.

This option is only valid in conjunction with MQGMO_LOCK. MQOO_BROWSE must have been included in the open options when the handle was opened.

MQGMO_UNLOCK

The message previously read with MQGMO_BROWSE_FIRST + MQGMO_LOCK is unlocked making it available to another application. If no message was previously locked then this call shall return with an error.

MQGMO_ACCEPT_TRUNCATED_MSG

The MQGET operation completes successfully, removing the message from the queue (at the syncpoint, if applicable), even though the BufferLength is shorter than the message (which would return a warning in the completion code). Without this option, a buffer which is too small causes the MQGET to complete unsuccessfully.

MQGMO_MSG_UNDER_CURSOR

The message previously read with MQGMO_BROWSE_FIRST + MQGMO_LOCK is deleted from the queue. If no message was previously locked then this call shall return with an error.

Notes:

1. This option must not be specified with the MQGMO_BROWSE_FIRST option. It is also an error if the queue was not opened both for browse and for input. If the browse cursor is not currently pointing to a retrievable message, an error is returned by the MQGET call.
2. If the MQGMO_WAIT option is specified with MQGMO_MSG_UNDER_CURSOR, it is ignored; no error is raised.

WaitInterval (MQLONG)

Wait interval.

The maximum time, expressed in milliseconds, that the MQGET call waits for a message to arrive. After this time, the call completes with an error (MQRC_NO_MSG_AVAILABLE).

This field is used in conjunction with the MQGMO_WAIT option. It is ignored if this is not specified.

The following special value is recognized:

MQWI_UNLIMITED

An unlimited wait is required.

Signal1 (MQLONG)

This is a reserved field.

Signal2 (MQLONG)

This is a reserved field.

ResolvedQName (MQCHAR48)

Resolved name of the destination queue.

This is an output field which is set by the queue manager to the local name of the queue from which the message was retrieved, as defined to the connected queue manager.

The resolved name is different from the name used to open the queue if an alias name was used. For the case of an alias queue, the name of the local queue is returned.

MQDLH - dead-letter header structure

StrucId (MQCHAR4)

Structure Identifier

The value must be:

MQDLH_STRUC_ID

Identifier for dead-letter header structure.

Version (MQLONG)

Structure version number.

The value must be:

MQDLH_VERSION_1

Version number for dead-letter header structure.

Reason (MQLONG)

Reason message arrived on dead-letter queue.

This identifies the reason why the message was placed on the dead-letter queue instead of on the original destination queue. It should either be one of the MQRC_* values (for example, MQRC_Q_FULL) or one of the following MQFB_* values:

- MQFB_DUPLICATE_MSG_SEQ_NUMBER

  For a requester or receiver channel, a message was received which contained a message sequence number for which a message has already been received. The message is put on the dead-letter queue.

- MQFB_ALSO_PUT_ON_REMOTE_DEAD_Q

  A message was transmitted along the channel, but the receiving end was unable to put the message onto the required queue. The receiving end has placed the message instead on the dead-letter queue at the receiving end, with a Reason code which identifies the problem. The message has also been placed on the dead-letter queue at the sending end, with a Reason of MQFB_ALSO_PUT_ON_REMOTE_DEAD_Q.

  Care should be taken with messages containing this feedback code, since another copy of the same message also exists at the receiving end of the channel. In order to avoid duplicating the effect of the message, only one of the two messages must be forwarded.

DestQName (MQCHAR48)

Name of original destination queue.

This is the name of the message queue that was the original destination for the message.

DestQMgrName (MQCHAR48)

Name of original destination queue manager.

This is the name of the queue manager that was the original destination for the message.

Encoding (MQLONG)

Original data encoding.

This specifies the data encoding used for numeric data in the original message. It applies to the message data which follows the MQDLH structure; it does not apply to numeric data in the MQDLH structure itself.

When an MQDLH structure is prefixed to the message data, the original data encoding should be preserved by copying it from the Encoding field in the message descriptor MQMD to the Encoding field in the MQDLH structure. The Encoding field in the message descriptor should then be set to the value appropriate to the numeric data in the MQDLH structure.

The value MQENC_NATIVE can be used for the Encoding field in both the MQDLH and MQMD structures.

CodedCharSetId (MQLONG)

Original coded character set identifier.

This specifies the coded character set identifier of character data in the original message. It applies to the message data which follows the MQDLH structure; it does not apply to character data in the MQDLH structure itself.

When an MQDLH structure is prefixed to the message data, the original coded character set identifier should be preserved by copying it from the CodedCharSetId field in the message descriptor MQMD to the CodedCharSetId field in the MQDLH structure. The CodedCharSetId field in the message descriptor should then be set to the value appropriate to the character data in the MQDLH structure.

The value MQCCSI_Q_MGR can be used for the CodedCharSetId field in the MQMD structure, but should not be used for the CodedCharSetId field in the MQDLH structure, as the queue manager does not replace the value MQCCSI_Q_MGR in the latter field by the value that applies to the queue manager.

Format (MQCHAR8)

Original format name.

This is the format name of the application data in the original message. It applies to the message data which follows the MQDLH structure; it does not apply to the MQDLH structure itself.

When an MQDLH structure is prefixed to the message data, the original format name should be preserved by copying it from the Format field in the message descriptor MQMD to the Format field in the MQDLH structure. The Format field in the message descriptor should then be set to the value MQFMT_DEAD_LETTER_HEADER.

PutApplType (MQLONG)

Type of application that put message on dead-letter queue.

This is the type of the application that decided to put the message on the dead-letter queue. This field has the same meaning as the PutApplType field in the message descriptor MQMD.

If it is the queue manager that redirects the message to the dead-letter queue, PutApplType has the value MQAT_QMGR.

PutApplName (MQCHAR28)

Name of application that put message on dead-letter queue.

This is the name of the application that decided to put the message on the dead-letter queue. The format of the name depends on the PutApplName field. See also, the description of the PutApplName field in MQMD.

If it is the queue manager that redirects the message to the dead-letter queue, PutApplName contains the first 28 characters of the queue-manager name, padded with blanks if necessary.

PutDate (MQCHAR8)

Date when message was put on dead-letter queue.

This is the date when the message was put on the dead-letter queue. The format that is used when this field is generated by the queue manager is:

YYYYMMDD

PutTime (MQCHAR8)

Time when message was put on dead-letter queue.

This is the time when the message was put on the dead-letter queue. The format that is used when this field is generated by the queue manager is:

HHMMSSTH

where the last two digits are tenths and hundredths of a second. The 24-hour clock is used, with a leading zero if the hour is less than 10. GMT is used (for this and for PutDate), subject to the accuracy of the system clock.

Table 21. Initial values of fields in MQDLH

Field Name	Name of Constant	Value of Constant
StrucId	MQDLH_STRUC_ID	DLH[a]
Version	MQDLH_VERSION_1	1
Reason	MQRC_NONE	0
DestQName	none	spaces
DestQMgrName	none	spaces
Encoding	none	0
CodedCharSetId	none	0
Format	none	spaces
PutApplType	MQAT_NO_CONTEXT	0
PutApplName	none	spaces
PutDate	none	spaces
PutTime	none	spaces

a. This must end in a single blank character

MQI return codes

For each MQI call, a completion code and a reason code are returned by the MQSeries System to indicate the success or failure of the MQI function. This section lists the possible codes.

MQI completion codes

The completion code (CompCode) parameter informs the application making an MQI call whether or not the call completed successfully, completed partially, or failed.

The possible completion codes are as follows:

0 MQCC_OK

Successful completion.

The call completed fully. All output parameters have been set.

The Reason parameter always has the value `MQRC_NONE` in this case.

1 MQCC_WARNING

Warning of partially completed call.

The call completed partially. Some output parameters may have been set in addition to the CompCode and Reason output parameters.

The Reason parameter gives additional information.

2 MQCC_FAILED

Call failed.

The processing of the call did not complete. The state of the queue manager is normally unchanged (exceptions are specifically noted). Only the CompCode and Reason output parameters have been set.

The reason may be a fault in the application program, or the reason may be a result of some situation outside the application, for example the application's authority may have been revoked.

The Reason parameter gives additional information.

MQI reason codes

The reason code (Reason) parameter is a qualification to the CompCode.

If there is no special reason to report, `MQRC_NONE` is returned. A successful call typically returns `MQCC_OK` and `MQRC_NONE`.

If the CompCode is either `MQCC_WARNING` or `MQCC_FAILED`, the queue manager always reports a qualifying reason. Details are provided under each call description.

An alphabetical listing of all reason codes and descriptions follows.

Note: Reason codes marked with an asterisk (*) are not currently implemented.

0 MQRC_NONE

No reason to report.

The call completed normally (CompCode is `MQCC_OK`).

Corrective action: None.

2000 *MQRC_ACCESS_RESTRICTED

Queue manager in restricted access mode.

The MQCONN call was rejected because the queue manager has been started in restricted access mode.

Corrective action: Contact your system administrator.

2001 MQRC_ALIAS_BASE_Q_TYPE_ERROR

Alias base queue not a valid type.

An MQOPEN or MQPUT1 request was issued, specifying an alias queue as the target, but the BaseQName in the alias queue attributed resolves to a queue that is not predefined local or remote queue.

Corrective action: Correct the queue definitions.

2002 MQRC_ALREADY_CONNECTED

Application already connected.

An MQCONN call was issued, but the application is already connected to the queue manager.

Corrective action: None. The Hconn parameter returned has the same value as was returned for the previous MQCONN call.

2004 MQRC_BUFFER_ERROR

Buffer parameter not valid.

Buffer is not valid. The parameter pointer is not valid, or points to read-only storage for MGET calls, or to storage that cannot be accessed for the entire length specified by BufferLength (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Correct the parameter.

2005 MQRC_BUFFER_LENGTH_ERROR

Buffer length parameter not valid.

BufferLength is not valid. The reason also occurs if the parameter pointer is not valid (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Specify a non-negative value.

2006 MQRC_CHAR_ATTRS_LENGTH_ERROR

Length of character attributes not valid.

CharAttrLength is negative (for MQINQ calls) or is not large enough to hold all selected attributes. This reason also occurs if the parameter pointer is not valid (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

2007 *MQRC_CHAR_ATTRS_ERROR

Character attributes string not valid.

CharAttrs is not valid. The parameter pointer is not valid, or points to read-only storage for MQINQ calls or to storage that is not as long as implied by CharAttrLength (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Correct the parameter.

2008 MQRC_CHAR_ATTRS_TOO_SHORT

Not enough space allowed for character attributes.

For MQINQ calls, CharAttrLength is not large enough to contain all of the character attributes for which MQCA_* selectors are specified in the Selectors parameter.

The call still completes, with the CharAttrs parameter string is filled in with as many character attributes as there is room for. Only complete attribute strings are returned. Space at the end of the string that is not large enough to hold the next attribute is unchanged.

Corrective action: Specify a large enough value, unless only a subset of the values is needed.

2009 MQRC_CONNECTION_BROKEN

Connection not established.

Connection to the queue manager has been lost or was not established. This can occur because the MQCONN call was not executed.

Corrective action: Applications must establish connection by issuing the MQCONN call.

2010 *MQRC_DATA_LENGTH_ERROR

Data length parameter not valid.

DataLength is not valid. The parameter pointer is not valid, or points to read-only storage (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Correct the parameter.

2013 MQRC_EXPIRY_ERROR

Expiry time not valid.

The Expiry field is reserved, and must have a value of -1.

Corrective action: Specify -1.

2014 MQRC_FEEDBACK_ERROR

Feedback code not valid.

A feedback code (Feedback) was specified in MQMD that is outside both the range defined for system feedback codes and that defined for application feedback codes.

Corrective action: Specify a valid value.

2016 MQRC_GET_INHIBITED

Gets failed for the queue.

MQGET calls are currently inhibited for this queue or for the queue that this queue resolves to.

Corrective action: Contact your system administrator.

2017 MQRC_HANDLE_NOT_AVAILABLE

No more handles available.

An MQOPEN or MQPUT1 request was issued, but the maximum number of open handles allowed for this task has already been reached.

Corrective action: Check whether the application is looping. Otherwise, reduce the complexity of the application. The maximum number of open handles that a task can have is queue-manager attribute (MaxHandles).

2018 MQRC_HCONN_ERROR

Connection handle not valid.

Hconn is not valid. This reason occurs if the parameter pointer is not valid, or points to read-only storage for the MQCONN call (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Ensure that a successful MQCONN call is performed for the queue manager instance, and that an MQDISC call has not already been performed for it. Check that the handle is being used within its valid scope. See the MQCONN call in this chapter.

2019 MQRC_HOBJ_ERROR

Object handle not valid.

Hobj is not valid. This reason also occurs if the supplied value is incorrect, the parameter pointer is not valid, or points to a read-only storage for an MQOPEN call (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Ensure that a successful MQOPEN call is performed for this object, and that an MQCLOSE call has not already been performed for it. For MQGET and MQPUT calls, also ensure that the handle represents a queue object. Check that the handle is being used within its valid scope. See the MQOPEN call in this chapter.

2021 MQRC_INT_ATTR_COUNT_ERROR

Count of integer attributes not valid.

IntAttrCount is negative (for MQINQ calls), or is not large enough to hold all selected attributes. This reason also occurs if the parameter pointer is not valid (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Specify a value large enough for all selected integer attributes.

2022 MQRC_INT_ATTR_COUNT_TOO_SMALL

Not enough space allowed for integer attributes.

For MQINQ calls, IntAttrCount is not as large as the number of integer attribute selectors (MQIA_*) specified in the Selectors parameter.

The call still completes, with the IntAttrs array filled with as many integer attributes as there is room for.

Corrective action: Specify a large enough value, unless only a subset of the values is needed.

2023 *MQRC_INT_ATTRS_ARRAY_ERROR

Integer attributes array not valid.

IntAttrs is not valid. The parameter pointer is not valid, or points to read-only storage for an MQINQ call or to storage that is not as long as indicated by IntAttrCount (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Correct the parameter.

2024 MQRC_LOCK_NOT_AVAILABLE

An internal MQI (Message Queuing Interface) error has occurred.

This error is implementation specific. Examine the error log for additional information. An MQOPEN, MQGET, MQPUT, or MQPUT1 request was issued, and it was necessary to acquire a lock, but an internal error occurred.

Corrective action: Get the error code from the error log if present and review the problem.

2025 MQRC_MAX_CONNS_LIMIT_REACHED

Connection initialization failure.

The MQCONN call was rejected because the initialization necessary to connect to the queue manager has failed.

Corrective action: Examine the error log for additional information.

2026 MQRC_MD_ERROR

Message descriptor not valid.

MQMD control block is not valid. Either the StrucId mnemonic eye-catcher is not valid, or the Version is not recognized. This reason also occurs if the parameter pointer is not valid, or points to read only storage (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Correct the definition of the message descriptor. Ensure that the required input fields are correctly set.

2027 MQRC_MISSING_REPLY_TO_Q

Missing reply-to-queue.

The reply-to-queue name (ReplyToQ) in MQMD is not specified (that is, it is all blanks), but a reply was requested (MQMT_REQUEST was specified in the MsgType field of the message descriptor).

Corrective action: Specify the name of the queue to which the reply is to be sent.

2029 MQRC_MSG_TYPE_ERROR

Message type in message descriptor not valid.

Message type (MsgType) in the message descriptor (MQMD) is not valid.

Corrective action: Ensure that a valid type is specified.

2030 MQRC_MSG_TOO_BIG_FOR_Q

Message length greater than maximum for queue.

An attempt was made to put a message that is bigger than allowed by the queue.

Corrective action: Check whether BufferLength was correctly specified. If so, either break the message into several smaller messages, or increase MaxMsgLength for the queue.

2031 MQRC_MSG_TOO_BIG_FOR_Q_MGR

This reason is not returned directly from an MQI call, but can occur in the Reason field of the MQDLH structure, if a message is put on the dead-letter queue.

This code is used if the message is put on the dead-letter queue because a channel, through which the message is to pass, has restricted the maximum message length to a value that is less than the message's length.

Note that the channel's maximum message length may be set to a value that is less than the maximum message length supported by the remote queue manager.

2033 MQRC_NO_MSG_AVAILABLE

No message available.

An MQGET call was issued, but there is no message on the queue that satisfies the criteria specified in the message descriptor. Either the MQGMO_WAIT option was not specified or it was specified, but the (non-zero) timeout interval has expired.

Corrective action: If this is an unexpected condition, check whether the message was successfully put on the queue.

Consider waiting longer for the message.

2034 *MQRC_NO_MSG_UNDER_CURSOR

A browse was not used before issuing MQGET with MQGMO_MSG_UNDER_CURSOR.

Corrective action: Check the error log for additional information.

2035 *MQRC_NOT_AUTHORIZED

Not authorized for access.

On the MQCONN call, the application is not authorized to connect to the queue manager. On MQOPEN or MQPUT1 calls, the application is not authorized to open the object for the option, or options, specified.

Corrective action: Ensure that the correct queue manager or object was specified, and that appropriate authority exists.

2036 MQRC_NOT_OPEN_FOR_BROWSE

Queue object not open for browse.

An MQGET call was issued to a queue not opened for browse with one of the following options:

- MQGMO_BROWSE_FIRST + MQGMO_LOCK
- MQGMO_MSG_UNDER_CURSOR

Corrective action: Specify MQOO_BROWSE when the queue is opened.

2037 MQRC_NOT_OPEN_FOR_INPUT

Queue object not open for input.

Corrective action: Specify MQOO_INPUT_EXCLUSIVE or MQOO_INPUT_SHARED when the queue is opened.

2038 MQRC_NOT_OPEN_FOR_INQUIRE

Queue object not open for inquire.

Corrective action: Specify MQOO_INQUIRE when the queue is opened.

2039 MQRC_NOT_OPEN_FOR_OUTPUT

Queue object not open for output.

Corrective action: Specify MQOO_OUTPUT when the queue is opened.

2041 *MQRC_OBJECT_CHANGED

Object definition changed since opened.

Since the Hobj handle used in this call was opened, object definitions that affect this object have been changed. See the MQOPEN call in this chapter.

Corrective action: Issue an MQCLOSE call to return the handle to the system. Reopen the object, obtaining a new handle, and retry the operation.

If object definitions are critical to the application logic, an MQINQ call can be used to find out what has changed. See the MQINQ call in this chapter.

2042 MQRC_OBJECT_IN_USE

Object already open with conflicting options.

An MQOPEN call has been issued, but the object in question has already been opened (by this or another application), with options that conflict with those specified in the Options parameter. This arises if the request is for shared input, but the object is already open for exclusive input, and also if the request is for exclusive input, but the object is already open for input.

Corrective action: System design should specify whether an application is to wait and retry, or take other action.

2043 MQRC_OBJECT_TYPE_ERROR

Object type not valid.

ObjectType (in MQOD) is not valid because the field specifies an unrecognized value. The object type must be MQOT_Q.

Corrective action: Specify a valid object type.

2044 MQRC_OD_ERROR

Object descriptor structure not valid.

MQOD control block is not valid. Either the StrucId mnemonic eye-catcher is not valid, or the Version is not recognized.

Corrective action: Correct the definition of the object descriptor. Ensure that required input fields are correctly set.

2045 MQRC_OPTION_NOT_VALID_FOR_TYPE

Option not valid for object type.

An attempt was made to open a transmission queue for output or a remote queue for input by an application.

Corrective action: Specify the correct option.

2046 MQRC_OPTIONS_ERROR

Options not valid or not consistent.

The Options field or parameter is unrecognized, or contains a combination that is not valid.

For MQGET, MQPUT, or MQPUT1 calls, this field is in the options structure (MQGMO or MQPMO) for the call.

This reason also occurs if the Options parameter pointer is not valid for MQOPEN or MQCLOSE calls (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results may occur).

Corrective action: Specify valid options. Check under the description of Options for the particular call, to see which option combinations are not valid.

2047 MQRC_PERSISTENCE_ERROR

Persistence not valid.

Persistence value in the message descriptor (MQMD) is not valid.

Corrective action: Specify a valid value.

2049 MQRC_PRIORITY_EXCEEDS_MAXIMUM

Priority not valid.

The specified priority is greater than 0.

Corrective action: None, this reason code is only a warning.

2050 MQRC_PRIORITY_ERROR

Priority not valid.

The Priority value in the message descriptor (MQMD) field is reserved, and must be specified as 0.

Corrective action: Specify 0 (zero).

2051 MQRC_PUT_INHIBITED

Puts inhibited for the queue.

MQPUT and MQPUT1 calls are currently inhibited for the queue(InhibitPut), or for the queue to which the alias queue resolves.

Corrective action: If the system design allows applications to inhibit put requests for short periods, retry the operation later.

2053 MQRC_Q_FULL

Queue already at maximum depth.

The MaxQDepth limit setting has been reached.

Corrective action: Retry the operation later. Consider increasing the maximum depth for the queue, or arranging for additional instances of the application servicing the queue.

2056 MQRC_Q_SPACE_NOT_AVAILABLE

No space available on disk for queue.

An MQPUT or MQPUT1 request was issued, but the request failed.

Corrective action: Review the error log for additional information.

2058 MQRC_Q_MGR_NAME_ERROR

Queue manager name not valid or not known.

The queue manager name specified for the MQCONN call is not valid. This reason also occurs if the parameter pointer is not valid (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results may occur).

Corrective action: Use an all-blank name if possible, or verify the name used is valid.

2059 MQRC_Q_MGR_NOT_AVAILABLE

Queue manager initialization failed.

Corrective action: Review the error log for additional information.

2061 MQRC_REPORT_OPTIONS_ERROR

Report options in message descriptor not valid.

The Report field in the message descriptor (MQMD) is not valid.

Corrective action: Set the field to 0 (zero).

2063 *MQRC_SECURITY_ERROR

Security error occurred.

The MQCONN call was rejected because a security error occurred.

Corrective action: Note the error from the security manager, and contact your system programmer.

2065 MQRC_SELECTOR_COUNT_ERROR

Count of selectors not valid.

The SelectorCount parameter specifies a value which is not valid. This reason also occurs if the parameter pointer is not valid (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results may occur).

Corrective action: Specify a value in the range 0 to 256.

2066 MQRC_SELECTOR_LIMIT_EXCEEDED

Count of selectors too big.

The SelectorCount parameter specifies a value larger than the maximum supported (256).

Corrective action: Reduce the number of selectors specified on the call. The valid range is 0 through 256.

2067 MQRC_SELECTOR_ERROR

Attribute selector not valid.

A selector in the Selectors array is not valid. This reason occurs if the parameter pointer is not valid (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results may occur).

Corrective action: Ensure that the value specified for the selector is valid for the object type represented by Hobj.

2068 MQRC_SELECTOR_NOT_FOR_TYPE

Selector not applicable for queue type.

On the MQINQ call a selector in the Selectors array is not applicable to the type of queue whose attributes are being queried.

The call still completes, with the corresponding element, or elements, of IntAttrs set to MQIAV_NOT_APPLICABLE for an integer attribute, or the appropriate portion, or portions, of the CharAttrs string set to a character string of all asterisks (*).

Corrective action: Check the value specified in the selector.

2069 MQRC_SIGNAL_OUTSTANDING

Signal outstanding for this handle.

An MQGET request was issued, with either the MQGMO_SET_SIGNAL or MQGMO_WAIT option, but there is already a signal outstanding for this object handle Hobj.

Corrective action: Check the application logic. If it is necessary to set a signal or wait when there is a signal outstanding for the same queue, a different object handle must be used.

2070 MQRC_SIGNAL_REQUEST_ACCEPTED

No message returned, but signal request was accepted.

An MQGET request was issued, specifying MQGMO_SET_SIGNAL in the GetMsgOpts parameter. No suitable message is currently available. The application can now wait on the Signal1 field.

Corrective action: Wait on the Signal1 field and when the signal is delivered, check this field to ensure that a message is now available. If it is, reissue the MQGET request.

2071 MQRC_STORAGE_NOT_AVAILABLE

Internal error.

Corrective action: Review the error log for additional information.

2072 MQRC_SYNCPOINT_NOT_AVAILABLE

Syncpoint was specified in the Options field (MQGMO_SYNCPOINT or MQPMO_SYNCPOINT) and is not supported.

Corrective action: Change the Options field to be MQGMO_NO_SYNCPOINT or MQPMO_NO_SYNCPOINT, as appropriate.

2079 MQRC_TRUNCATED_MSG_ACCEPTED

Truncated message returned (message deleted from queue).

On an MQGET call, the message length was too large to fit in the supplied buffer. MQGMO_ACCEPT_TRUNCATED_MSG was specified, so the call completes. The message is removed from the queue, or, if this was a browse operation, the browse cursor advanced to this message.

The DataLength field is set by the system, and Buffer contains as much of the message as fits.

Corrective action: None, because the application expected this situation.

2080 MQRC_TRUNCATED_MSG_FAILED

Truncated message returned (message not deleted from queue).

On an MQGET call, the message length was too large to fit in the supplied buffer. MQGMO_ACCEPT_TRUNCATED_MSG was not specified, so the message is not removed from the queue, or, if this was a browse operation, the browse cursor remains where it was before this call.

The DataLength field is set by the system, and Buffer contains as much of the message as fits.

Corrective action: Supply a large enough buffer, or specify MQGMO_ACCEPT_TRUNCATED_MSG if not all of the message data is required.

2082 *MQRC_UNKNOWN_ALIAS_BASE_Q

Unknown alias base queue.

An MQOPEN or MQPUT1 request was issued, specifying an alias queue as the target, but the BaseQName in the alias queue attributes is not recognized as a queue name.

Corrective action: Correct the queue definitions.

2085 MQRC_UNKNOWN_OBJECT_NAME

Unknown object name.

The ObjectName in the object descriptor (MQOD) is not recognized for the specified object type.

Corrective action: Specify a valid object name. Ensure that the name is padded to the right with blanks if necessary.

2086 MQRC_UNKNOWN_OBJECT_Q_MGR

Unknown object queue manager.

The ObjectQMgrName in the object descriptor (MQOD) is not valid for MQOPEN nor MQPUT1.

Corrective action: Specify a valid queue manager name (or all blanks or an initial null character to refer to the connected queue manager instance). Ensure that the name is padded to the right with blanks if necessary. A transmission queue should normally be defined for each remote queue manager to which messages can be sent.

2087 MQRC_UNKNOWN_REMOTE_Q_MGR

Unknown remote queue manager.

An MQOPEN or MQPUT1 request was issued, specifying a remote queue as the target, but the RemoteQMgrName in the remote queue attributes is not recognized as valid.

Corrective action: Correct the queue definitions. Ensure that a transmission queue is defined for each remote queue manager to which messages can be sent.

2090 MQRC_WAIT_INTERVAL_ERROR

Negative wait interval in MQGMO.

A negative time-out (WaitInterval) value was specified in MQGMO (other than the special value MQWI_UNLIMITED).

Corrective action: Specify a value greater than or equal to zero, or MQWI_UNLIMITED.

2091 MQRC_XMIT_Q_TYPE_ERROR

Transmission queue not local.

On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. There is a queue defined on the connected queue manager with the same name as the remote queue manager, but this is not a local queue.

Corrective action: If a non-blank ObjectQMgrName was specified in the ObjDesc parameter, ensure that it was correct. Otherwise, correct the queue definitions.

2092 MQRC_XMIT_Q_USAGE_ERROR

Transmission queue with wrong usage.

On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. There is a local queue defined on the connected queue manager with the same name as the remote queue manager, but the local queue does not have a Usage of MQUS_TRANSMISSION.

Corrective action: Correct the queue definition.

2173 MQRC_PMO_ERROR

Put-message options structure not valid.

On an MQPUT or MQPUT1 call, the MQPMO structure is not valid. Either the StrucId mnemonic eye-catcher is not valid, or the Version is not recognized. This reason also occurs if the parameter is not valid, or points to read-only storage (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Correct the definition of the MQPMO structure. Ensure the required input fields are correctly set.

2186 MQRC_GMO_ERROR

Get-message options structure not valid.

On an MQGET call, the MQGMO structure is not valid. Either the *StrucId* mnemonic eye-catcher is not valid, or the *version* is not recognized. This reason also occurs if the parameter is not valid, or points to read-only storage (it is not always possible to detect parameter pointers which are not valid; if it is not detected, unpredictable results occur).

Corrective action: Ensure that the MQGMO structure is initialized and correctly passed on the MQGET call.

2195 MQRC_UNEXPECTED_ERROR

Unexpected error occurred.

Corrective action: Contact your system programmer.

2196 MQRC_UNKNOWN_XMIT_Q

Transmission queue specified by a remote queue definition no longer exists.

Corrective action: Contact your system programmer.

2206 MQRC_MSG_ID_ERROR

MsgId field is invalid. This field must be MQMI_NONE for MQGETs.

2207 MQRC_CORREL_ID_ERROR

CorrelId is invalid.

For MQGETs, this field must be set to MQCI_NONE.

2209 MQRC_NO_MSG_LOCKED

An MQGET call was issued with the MQGMO_UNLOCK option while no lock was being held for that Hobj.

Corrective action: Review application logic for possible prior error that caused a lock to not be held (that is, no more messages).

Appendix A. UnixWare error messages

MQSeries System internal messages

Understanding MQSeries System internal messages

This appendix contains an abridged listing of the IBM MQSeries for UnixWare internal error messages. More detailed error message information may be found in *IBM MQSeries Version 1 Products for UNIX Operating Systems Messages and Codes*, SC33-1754. Each message contains the date, time, MQSeries System Function and the MQSeries System Message. The format for the Internal MQSeries System Messages is as follows:

Example Message:

day-mth-yr hr:min.sec <MQSeries_System_function> <MQSeries_System_message>

Message Definition:

day-mth-yr	Represents the current date in a two character day, three character month and two character year.
hr:min.sec.	Represents the current time in hours, minutes and seconds.
MQSeries_System_function	MQSeries System function name within the system.
MQSeries_System_message	MQSeries System message text defining the error or message.

These MQSeries System Functions and MQSeries System Messages are presented in alphabetical order. A MQSeries System Function can be associated with several MQSeries System Messages. These messages can also be used with other MQSeries System Functions. Parameters, notations, date and time stamps contained in the text of a message will be shown within <> marks.

Accompanying each message and code is the following information, when applicable:

- **Explanation**: This section tells what the message or code means, why it occurred and what caused it.
- **Module**: This section indicates which modules issued the message, to assist in diagnosing problems.
- **Severity**: Severity values have the following meanings:

0	An information message. No error has occurred.
4	A warning message. A condition has been detected of which the user should be aware. The user may need to take further action.
8	An error message. An error has been detected and processing could not continue.
12	A severe error message. a severe error has been detected and processing could not continue.

- **System action**: This part tells what is happening as a result of the condition causing the message or code. If this information is not shown, no system action is taken.
- **User response**: If a response by the user is necessary, this section tells what the appropriate responses are, and what their effect is. If this information is not shown, no user response is required.
- **Operator response**: If an operator response is necessary, this section tells what the appropriate responses are, and what their effect is. If this information is not shown, no operator response is required.
- **System programmer response**: If a response by the system programmer is required, this part tells what the appropriate responses are, and what their effect is. If this information is not shown, no system programmer response is required.
- **Programmer response**: If a programmer response is necessary, this part tells what the appropriate responses are, and what their effect is. If this information is not shown, no programmer response is required.
- **Problem determination**: This section lists the actions that can be performed to obtain adequate data for support personnel to diagnose the cause of the error. If this information is not shown, no problem determination is required.

Internal MQSeries System function names

AddQueue
AttachMgr
AttachObj
CloseIndex
DeleteQueue
DeleteWin
DetachMgr
DetachObj
ezMgrClose
ezObjClose
ezObjCreate
ezObjDelete
ezObjInq
ezObjOpen
ezObjRead
ezObjUnlock
ezObjWrite
ezQueDelete
ezQueReorg
ezRecoverMSN
fCountUsers
fOpenQULFile
FindQMGR
GetDefQMGR
LoadEnvironment
LoadQDT
LockIndex
LockRecord
MapIBMCode
ModifyQueueDefinition
MQCLOSE
MQCONN
MQDISC
MQGET
MQINQ
MQOPEN
MQPUT
MQPUT1

NewWin
OpenIndex
OpenQDT
OpenQMGR
OpenQueueManager
PrintWinList
ReadIndex
ReadQDT
ReadQMGR
ReadQueueManager
RecoverInboundMSN
RecoverOutboundMSN
ScanIndex
SetDefQMGR
SetIndex
UnlockRecord
WriteIndex
WriteQDT
WriteQMGR
WriteQueueManager

MQSeries System messages

```
Could not allocate <number> bytes memory <sys error number>
Could not find mapping for <48 character string>/<48 character string>
Could not lock record <number> <sys error number>
Could not open <string> <sys error number>
Could not open index file <sys error number>
Could not open Queue Manager Database <sys error number>
Could not read default QueueManager <sys error number>
Could not rename file <string> <sys error number>
Could not seek to offset <number> <sys error number>
Could not unlink file <string> <sys error number>
Could not unlock record <number>. <sys error number>
Could not write default QueueManager <sys error number>
Failed closeing queue <string> <sys error number>
Failed opening QDT <string> <sys error number>
Failed opening queue <string> <sys error number>
Failed reading index <sys error number>
Failed seeking index <sys error number>
Failed seeking record <number> <sys error number>
Failed to close index <sys error number>
Failed to locate index for <string>
Failed to lock index <sys error number>
Failed to unlock index <sys error number>
Failed writing <number> bytes to QDT <sys error number>
Failed writing <number> of <number> bytes to QMGR <sys error number>
Found a new MSN: <number> for <string>
Index <string> already exists
Invalid record index passed <number> vs. <number>
New window created for <number>
MCAMD rejected message, reason: <text>
 where <text> is one of the following:
      "No error"
      "See error log"
      "Non-Specific reason"
      "Channel already exists"
      "Channel put record failed"
      "Channel get record failed"
      "Channel delete record failed"
```

```
        “No valid ChannelList index”
        “Channel reserved by other MQM or MCA”
        “Channel in use”
        “Connection in improper state”
        “In control of another channel”
        “Not in control of any channel”
        “Unable to send msg to MCA”
        “MCA already stopped”
        “MCA already running”
        “Message type undefined”
        “Connection in improper state”
        “QMgr index exceeded maximum”
        “Unable to locate QMgr record”
        “Unable to create MCA shared memory”
        “Invalid message length”
        “Exceed maximum client count”
        “Invalid message type”
        “Matching pid not found”
        “Unknown reason code”
No available slots in index table
No messages were copied
Opening <string>
Operation failed, Queue in use
Process not authorized for GET's from <48 character string>#<48 character string>
Queue <specific string length> is in use. ReOrganization aborted!
Queue <specific string length> successfully ReOrganized
Queue manager default record corrupt!
Queue manager not properly defined
Read failed! Got <number> of <number> <sys error number>
Read failed. Read <number> of <number> bytes from <string> <sys error number>
Read of QMGR failed! Read <number> of <number> <sys error number>
Record not written with proper version of MQI
Window deleted for <number>
Win entry for <number>
Write failed. Wrote <number> of <number> bytes to <string> <sys error number>
Write to index file failed <sys error number>
Wrong reply size from MCAMD. Rcvd <number> bytes, expected <number> bytes.
<number> byte <string> removed from the beginning of the file.
<number> byte <string> representing <number> <number>-byte record <string> were
copied.
<number> byte <string> truncated from the end of the file.
```

MCA error messages

Understanding MCA error messages

This appendix contains the IBM MQSeries for UnixWare internal error messages. Each message contains the date, time, MCA Function and the MCA Message. The format for the Internal MCA Messages is as follows:

Example Message:

```
day-mth-yr hr:min.sec <mca_function> <mca_message>
```

Message Definition:

`day-mth-yr`	Represents the current date in a two character day, three character month and two character year.
`hr:mn:sc.mls`	Represents the current time in hours, minutes, seconds and milliseconds.
`mca_function`	MCA function name within the system.

`mca_message` MCA message text defining the error or message. Although not shown, for most messages, the name of the affected channel appears at the beginning of the text.

These MCA Functions and MCA Messages are presented in alphabetical order. A MCA Function can be associated with several MCA Messages. These messages can also be used with other MCA Functions. Parameters, notations, date and time stamps contained in the text of a message will be shown within <> marks.

Internal MCA function names

CloseNamedPipe
CloseSocket
CreateMCAMem
CreateQMgrMem
DelChannelRec
DelMCAMDClient
dump_rcvd_msg
dump_xmit_msg
get_channel_name_list
GetChannelRec
InitChannelCB
main
mca_abort_msg
mca_ack_rcvd
mca_alloc_channel
mca_check_init_reply
mca_check_shutdown
mca_check_stop
mca_check_tsh
mca_chkpt_channel
mca_data_rcvd
mca_delete_msg
mca_deque_msg
mca_exit
mca_init_data_rejected
mca_init_dl_queue
mca_init_line
mca_init_mcamd_conn
mca_init_mem
mca_init_qmgr
mca_init_queues
mca_init_rcvd
mca_init_signals
mca_log_nak
mca_mcamd_cmd
mca_mcamd_init_cmpl
mca_msg_rcvd
mca_no_msg
mca_output_format
mca_queue_msg_dlq
mca_reject_init_data
mca_remove_msg
mca_reset_rcvd
mca_resync_rcvd
mca_send_confirmation
mca_sna_shutdown
mca_status_rcvd
mca_trace
mca_trace_input

mca_trace_output
mca_trace_rcvd_ack
mca_trace_send_ack
mca_wait_for_cmd
mca_wait_to_reset
MCAmain
MCAMDCloseConn
MCAMDConnect
MCAMDControl
MCAMDCreateChannel
MCAMDDeleteChannel
MCAMDListChannels
MCAMDmain
MCAMDModifyChannel
MCAMDOpenConn
MCAMDQueryChannel
MCAMDReleaseControl
MCAMDReserveControl
MCAMDStartChannel
MCAMDStartTrace
MCAMDStopChannel
MCAMDStopTrace
MCAMDUpdateChannel
mcautils
parse_command
PutChannelRec
ReadClientMsg
RecoverQueues
SendCLReply
SendLogonReply
SendMCAMsg
SendQueryReply
SendStatus
ServMCAClient
ServMCAOPERClient
soc_accept
soc_close
soc_conn
soc_open
soc_sub_read

MCA messages

```
Attempt to perform connect to MCAMD socket failed, error = <number>
Error executing deallocate, error = <number>
Error executing receive, error = <number>
Failed to write to the MCAMD socket
Failure during logon attempt to MCAMD
MCAMD process refused socket connection
Received client message with invalid client type
Received improper message from MCAMD process
Attempt to create <string> shared memory segment failed
Binary encoding is being negotiated
CCSID is being negotiated
Channel <channel number>, State <string>, Event Code <number>
Channel name exceeds maximum length
Channel stopped
Clients message size does not match message type, expected <number> and received
<number>
Close channel flag set
```

```
Connection to MCAMD complete
Disconnect timer has expired, disconnecting
Error allocating Channel
Error allocating memory for queue header
Error attaching to application queue, reason = <number>
Error attaching to queue, reason = <number>
Error attempting to dequeue a message from transmission queue, Completion code =
<number>,reason = <number>
Error closing client process socket
Error creating shared memory key
Error deleting message from transmission queue, Completion code = <number>, reason =
<number>
Error destroying shared memory segment for MCA client process
Error establishing the LU62 connection, error = <number>
Error executing allocate connection, error = <number>
Error executing attach connection, error = <number>
Error executing flush, error = <number>
Error executing get send state, error = <number>
Error executing gettimer, error = <number>
Error executing incinterval, error = <number>
Error executing prepare to receive, error = <number>
Error executing queue inquire, completion code = <number>, reason = <number>
Error executing request to send, error = <number>
Error executing send, error = <number>
Error executing shared memory attach
Error executing shared memory get
Error flag in ACK set, flags = <number>
Error in line code, error = <number>
Error incrementing message sequence number
Error opening Channel Name List file, error = <number>
Error opening queue, Completion code = <number>, reason = <number>
Error opening queue, completion code = <number>, reason = <number>
Error processing Channel List File
Error reading Channel Database file, error = <number>
Error reading client process message, error = <number>
Error setting file pointer in Channel Database file, error = <number>
Error unlocking message send in batch, Completion code = <number>, reason = <number>
Error writing Channel Database file, error = <number>
Error writing channel list reply message to client process
Error writing logon reply message to client process
Error writing message to client MCA process
Error writing query reply message to client process
Error writing status message to client process
Failed to read from the MCAMD socket, error = <number>
Initialization complete
Internal error calling system subroutine select, error = <number>
INVALID EVENT; Channel <channel number>, State <string>, Event Code <number>
Line initialized
Logical Unit of Work Identification (LUWID) mismatch
Maximum size of message in negotiation
Maximum size of transmitted message in negotiation
mca -c ChannelName -f ChannelListFile -m QueueMgrName
Message Channel Agent program exiting
Message Channel Agent Starting, PID = <number>
Number of Channels requested exceeds capability
rcv stat, rtn <number>, errno <number>, what_data_rcvd <number>, w_ctrl_rcvd
<number>,rq_t_snd <number>
Received a undefined client message
Reconnect retry limit exceeded
Request close flag set
```

```
Sequence number mismatch in negotiation
Size of message batch in error
Trace input
Trace output
Trace received ack
Trace sent ack
Unable to access Queue Manager Database file, error = <number>
Unable to allocate memory for Channel buffer
Unable to allocate memory for Channel control block, error = <number>
Unable to connect to Queue Manager, completion code = <number>, reason = <number>
Unable to connect to the MCAMD process
Unable to initialize signal handlers
Unable to initialize transport connection
Unable to locate database records for all channels
Unable to open all transmission queues
Unable to open Channel Database file, error = <number>
Wait for command
Wait for command
```

Transport layer protocol for LU 6.2 functions

```
tlp_init_cpic
tlp_rqst_send_cpic
tlp_rcv_cpic
tlp_send_cpic
tlp_alloc_cpic
tlp_attach_cpic
tlp_deallocate_cpic
tlp_abend_cpic
tlp_turn_line_cpic
```

Transport layer protocol for LU 6.2 CPI-C messages

In the following messages, the return codes are standard CPI-C return codes, which can be looked up in the EXPRESS Common Programming Interface for Communications (CPI-C) Programmer's Guide, Appendix E, `cpic.h` Include File or in the file /usr/express/include/cpic.h. If any log message shows a return code of 20, an additional log message will show the Express product-specific return code, which can also be found in the above manual and file.

```
<channel_name>: cminit failed, return code = <cpic_rc>
<channel_name>: cminic failed, return code = <cpic_rc>
<channel_name>: cmsmn failed, return code = <cpic_rc>
<channel_name>: cmspln failed, return code = <cpic_rc>
<channel_name>: cmssln failed, return code = <cpic_rc>
<channel_name>: cmstpn failed, return code = <cpic_rc>
<channel_name>: cmallc failed, return code = <cpic_rc>
<channel_name>: cmflus failed, return code = <cpic_rc>
<channel_name>: cmsltp failed, return code = <cpic_rc>
<channel_name>: cmspm failed, return code = <cpic_rc>
<channel_name>: xcsdq failed, return code = <cpic_rc>
<channel_name>: cmacci failed, return code = <cpic_rc>
<channel_name>: xcecfd failed, return code = <cpic_rc>
<channel_name>: xctest failed, return code = <cpic_rc>
<channel_name>: xctest failed, return code = <cpic_rc>, conv_return_code = <cpic_rc>
<channel_name>: cmrts failed, return code = <cpic_rc>
rcv param: fd <number>, buff <number>, length <number>
<channel_name>: cmrcv failed, return code = <cpic_rc>
Extract product-specific return code: rc = <cpic_rc>, specific_rc = <cpic_rc>
```

Internal transport layer protocol for TCP/IP functions

tlp_accept_tcpip

tlp_close_tcpip

tlp_conn_tcpip

tlp_init_tcpip

tlp_open_tcpip

tlp_recv_tcpip

tlp_send_tcpip

Internal transport layer protocol for TCP/IP messages

```
socket read timed out
Error closing socket, error = <number>
Error creating socket, error = <number>
Error getting port number for service <name> by name (getservbyname)
Error executing socket accept, error = <number>
Error executing socket bind, error = <number>
Error executing socket connect, error = <number>
Error executing socket listen, error = <number>
Error getting host name entry (gethostbyname) <name>
Error reading socket, error = <number>
Error unlinking socket <name>, error = <number>
<channel name>: Error executing close, error = <number>
<channel name>: Error executing send, error = <number>
<channel name>: Error setting socket options, error = <number>
<channel name>: Error attempting to accept connection on internet socket, error =
<number>
<channel name>: Unable to connect to internet socket at host <name>, service <name>
<channel name>: Error attempting to create listener socket, error = <number>
<channel name>: Error attempting to receive message on internet socket, error =
<number>
<channel name>: Remote channel disconnected from socket connection
Channel <channel name> received TSH with message length in excess of maximum
transmission length (<number> vs <number>)
```

Internal MCA daemon functions

CloseSocket
CreateMCAMem
CreateQMgrMem
DelChannelRec
DelMCAMDClient
GetChannelRec
InitChannelCB
MCAMDCloseConn
MCAMDControl
MCAMDCreateChannel
MCAMDDeleteChannel
MCAMDListChannels
MCAMDmain
MCAMDModifyChannel
MCAMDOpenConn
MCAMDQueryChannel
MCAMDReleaseControl
MCAMDReserveControl
MCAMDStartChannel
MCAMDStartTrace
MCAMDStopChannel

MCAMDStopTrace
MCAMDUpdateChannel
parse_command
PutChannelRec
ReadClientMsg
RecoverQueues
SendCLReply
SendLogonReply
SendMCAMsg
SendQueryReply
SendStatus
SendStatus
ServMCAClient
ServMCAOPERClient

Internal MCA daemon messages

Channel <channel number> not found
Channel database checkpoint failed, unable to update Channel <channel number>, error = <number>
Clients message size does not match message type, expected <number> and received <number>
Error accepting Internet operator connection, error = <number>
Error accepting Unix MCA connection, error = <number>
Error accepting Unix operator connection, error = <number>
Error adding Internet operator client
Error adding Unix MCA client
Error adding Unix operator client
Error closing client process socket
Error creating shared memory key, error = <number>
Error destroying shared memory segment for MCA client process
Error executing shared memory attach, error = <number>
Error executing shared memory get, error = <number>
Error opening Internet operator socket, error = <number>
Error opening Unix MCA socket, error = <number>
Error opening Unix operator socket, error = <number>
Error reading Channel Database file, error = <number>
Error reading client process message, error = <number>
Error recovering queues, Get record failed for Channel <channel number>
Error recovering queues, Put record failed for Channel <channel number>
Error setting file pointer in Channel Database file to record = <number>, error = <number>
Error writing Channel Database file, error = <number>
Error writing Channel list reply message to client process
Error writing logon reply message to client process
Error writing message to client MCA process
Error writing query reply message to client process
Error writing status message to client process
Internal error calling system subroutine select, error = <number>
Invalid command received
Message Channel Agent Maintenance Daemon Starting, PID = <number>
Message Channel Agent Maintenance Daemon Stopping, PID = <number>
Received an undefined client message
Received client message with invalid client type
Unable to access Queue Manager Database file for Queue Manager <queue name>
Unable to locate Channel Database record for Channel <channel number>
Unable to locate database records for all Channels
Unable to open Channel Database file, error = <number>

Appendix B. Sample source listings

zmqecho.c

```
/*************************************************************
*  Licensed Materials - Property of IBM                      *
*                                                            *
*  5765-519                                                  *
*  5765-521                                                  *
*  63H9503,5697-265                                          *
*  5765-520                                                  *
*  5765-516                                                  *
*  5765-513                                                  *
*  (C) Copyright IBM Corp. 1993, 1997                        *
*                                                            *
*  US Government Users Restricted Rights - Use,              *
*  duplication or disclosure restricted by GSA ADP           *
*  Schedule Contract with IBM Corp.                          *
*************************************************************/
/*************************************************************
 *
 *  Filename:       zmqecho.c
 *  Creation Date:  04-Mar-93
 *  Sccs Revision:  "@(#)zmqecho.c IBM Version 1.4 Revised
 *                                              11/14/94"
 *  Overview:
 *           This program will read a specified number of messages
 *            from a source queue, and write them to a destination
 *           queue.
 *
 *  Modification History:
 *  Date    Author  Reason
 *  ~~~~    ~~~~~~  ~~~~~~
 *  04-Mar-93  Originate.
 *  20-May-94   When echoing messages received from a remote
 *              system some fields may be set to invalid values
 *              (maintained solely for the use of the receiving
 *              application). These values must be set to valid
 *              values before they can be put onto the
 *              destination.
 *
 *************************************************************/

#include    <stdio.h>
#include    <string.h>
#include    <time.h>
#include    <stdlib.h>
#include    <sys/timeb.h>
#include    "cmqc.h"

#define     NAME_DELIMETER  '#'
#define     USAGE   \
"\n\007Usage: %s <# msgs> <Src QMGR>#<Src Que> <Dst QMGR>#<Dst
                        Que>\n\n"

static char SccsID[] = "@(#)zmqecho.c   IBM Version 1.4 Revised
                        11/14/94";
char    *ProgramName;

int main(int argc, char *argv[])
{
/*
*  For the MQCONN call, we shall connect to the default queue
*  manager.  This queue manager connection is used for all
*  calls.  Of course though, the queue manager names specified
*  on the command line are used in our MQOPEN calls.
*/
    MQCHAR48 QueueManager = "   "; /* Use default queue manager */
    MQHCONN qm_handle;

/*
*  The following variables deal with the source queue.
*/
    MQCHAR48 source_queue_manager, /*Name of source QueueManager*/
                source_queue; /* Name of source queue */
    MQHOBJ      source_que_handle; /*Handle to source queue*/
    MQGMO       get_options = { MQGMO_DEFAULT };
    MQOD        source_obj_desc = { MQOD_DEFAULT };

/*
*  The following variables deal with the destination queue.
*/
    MQCHAR48    dest_queue_manager, /* Name of destination QM */
                dest_queue; /* Name of destination queue */
    MQHOBJ      dest_que_handle;
    MQPMO       put_options = { MQPMO_DEFAULT };
    MQOD        dest_obj_desc = { MQOD_DEFAULT };

/*
*  The following variables are common to all MQI routines.
*/
    MQMD        msg_desc = { MQMD_DEFAULT };
    MQLONG      comp_code = MQCC_OK,
                reason = MQRC_NONE;

/*
* The following variable is used in call to MQINQ. We need to call
* this in order to determine how large of a buffer we need to
* allocate for reading a message from the source queue.
*/
    MQLONG      selector = MQIA_MAX_MSG_LENGTH;

    char      *data_buffer; /* Pointer to data buffer */
    long      buffer_length, /* Length of buffer (MQINQ) */
              messages_to_echo, /*Number of messages to copy*/
              message_length, /* Length of message to copy */
              read_index, /* Message being copied */
              name_length = 0, /* Length of arg name parsed */
              arg_pos = 0; /* Position within curr arg */

    ProgramName = *argv; /* Save program name for errors */

    if (argc != 4)
    {
        fprintf(stderr, USAGE, ProgramName);
        exit(1);
    }

/*
* Get number of messages.
*/
    messages_to_echo = atoi(argv[1]);

/*
* Grab Source queue manager queue name combination.
*/
    name_length = 0;
    while ( argv[2][arg_pos]
                && arg_pos < MQ_Q_MGR_NAME_LENGTH
                && argv[2][arg_pos] != NAME_DELIMETER)
    {
        source_queue_manager[name_length++] = argv[2][arg_pos++];
    }
    if (arg_pos < MQ_Q_MGR_NAME_LENGTH)
     memset((void *)&source_queue_manager[name_length], (int)' ',
               (size_t)(MQ_Q_MGR_NAME_LENGTH - name_length));
    strcpy(source_queue, &argv[2][++arg_pos]);
    if (strlen(source_queue) < MQ_Q_NAME_LENGTH)
    memset((void *)&source_queue[strlen(source_queue)], (int)' ',
           (size_t)(MQ_Q_NAME_LENGTH - strlen(source_queue)));
```

```
    strncpy(source_obj_desc.ObjectName, source_queue,
          sizeof(MQCHAR48));
    strncpy(source_obj_desc.ObjectQMgrName, source_queue_manager,
                                    sizeof(MQCHAR48));

/*
* Grab destination queue manager queue name combination.
*/
    arg_pos = 0;
    name_length = 0;
    while ( argv[3][arg_pos]
                && arg_pos < MQ_Q_MGR_NAME_LENGTH
                && argv[3][arg_pos] != NAME_DELIMETER)
    {
        dest_queue_manager[name_length++] = argv[3][arg_pos++];
    }
    if (arg_pos < MQ_Q_MGR_NAME_LENGTH)
       memset((void *)&dest_queue_manager[name_length], (int)' ',
                        (size_t)(MQ_Q_MGR_NAME_LENGTH -
                          name_length));
    strcpy(dest_queue, &argv[3][++arg_pos]);
    if (strlen(dest_queue) < MQ_Q_NAME_LENGTH)
        memset((void *)&dest_queue[strlen(dest_queue)], (int)' ',
                        (size_t)(MQ_Q_NAME_LENGTH -
                          strlen(dest_queue)));

    strncpy(dest_obj_desc.ObjectName, dest_queue,
                          sizeof(MQCHAR48));
    strncpy(dest_obj_desc.ObjectQMgrName, dest_queue_manager,
                                    sizeof(MQCHAR48));

/*
* Access and write messages to queue.
*/
    MQCONN(QueueManager, &qm_handle, &comp_code, &reason);
    if (comp_code)
    {
       printf("Failed to connect to the queue manager.  (Reason =
                          %ld)\n",
                         reason);
        return(1);
    }

    MQOPEN(qm_handle, &source_obj_desc, MQOO_INPUT_SHARED |
                          MQOO_INQUIRE,
                    &source_que_handle, &comp_code, &reason);
    if (comp_code)
    {
       printf("Failed to attach to source queue. Reason = %ld\n",
                          reason);
        printf("%.-48s\n%-.48s\n", source_obj_desc.ObjectName,
                    source_obj_desc.ObjectQMgrName);
        return(1);
    }

    MQINQ(qm_handle, source_que_handle, 1, &selector, 1,
                          &buffer_length, 0,
                NULL, &comp_code, &reason);
    if (comp_code)
    {
        printf("%s: Failed to get queue size.  (Reason = %ld)\n",
                        ProgramName, reason);
        MQCLOSE(qm_handle, &source_que_handle, MQCO_NONE,
                          &comp_code, &reason);
        MQCLOSE(qm_handle, &dest_que_handle, MQCO_NONE,
                          &comp_code, &reason);
        MQDISC(&qm_handle, &comp_code, &reason);
        return(1);
    }
/*
*  Based on value returned from MQINQ, allocate buffer for data
*  reads.
*/
    data_buffer = (char *)malloc((size_t)buffer_length);

    MQOPEN(qm_handle, &dest_obj_desc, MQOO_OUTPUT | MQOO_INQUIRE,
                    &dest_que_handle, &comp_code, &reason);
    if (comp_code)
    {
        printf("Failed to attach to destination queue. Reason =
                          %ld\n", reason);
        printf("%.-48s\n%-.48s\n", dest_obj_desc.ObjectName,
                    dest_obj_desc.ObjectQMgrName);
        return(1);
    }

    for (read_index = 0; read_index < messages_to_echo;
                          ++read_index)
    {

        memcpy(msg_desc.MsgId, MQMI_NONE, sizeof(MQBYTE24));
        memcpy(msg_desc.CorrelId, MQCI_NONE, sizeof(MQBYTE24));
/*
*  Perform destructive read of data.
*/
        MQGET(qm_handle, source_que_handle, &msg_desc,
                          &get_options,
                            buffer_length, data_buffer,
                          &message_length,
                                    &comp_code, &reason);
        if (comp_code)
        {
            if (reason == MQRC_NO_MSG_AVAILABLE)
            {
                printf("%s: No messages left to read\n",
                          ProgramName);
            }
            else
            {
                printf("Failed to get message #%ld. (Reason =
                          %ld)\n",
                                        read_index, reason);
            }
            break;
        }

/*
* 20-May-94
* At this point certain fields must be modified to legal values.
*/
        msg_desc.Report = MQRO_NONE;
        msg_desc.Persistence = MQPER_PERSISTENCE_AS_Q_DEF;
        msg_desc.Priority = 0L;

/*
*  Write data just read to the destination queue.
*/
       MQPUT(qm_handle, dest_que_handle, &msg_desc, &put_options,
                          message_length, data_buffer, &comp_code,
                                &reason);
        if (comp_code)
        {
           printf("Failed to put message #%ld. (Reason = %ld)\n",
                                        read_index, reason);
            break;
        }
    }

    free(data_buffer);

/*
*  Close source queue.
*/
    MQCLOSE(qm_handle, &source_que_handle, MQCO_NONE, &comp_code,
                          &reason);
    if (comp_code)
    {
        printf("Failed to close queue.  (Reason = %ld)\n",
                          reason);
```

```
    }

/*
*  Close destination queue.
*/
    MQCLOSE(qm_handle, &dest_que_handle, MQCO_NONE, &comp_code,
                          &reason);
    if (comp_code)
    {
       printf("Failed to close queue.  (Reason = %ld)\n", reason);
    }

/*
*  Disconnect.
*/
    MQDISC(&qm_handle, &comp_code, &reason);
    if (comp_code)
    {
        printf("Failed to disconnect from the queue manager.
                          (Reason = %ld)\n",
                                reason);
    }
}
```

zmqread.c

```
/*****************************************************
*  Licensed Materials - Property of IBM                *
*                                                      *
*  5765-519                                            *
*  5765-521                                            *
*  63H9503,5697-265                                    *
*  5765-520                                            *
*  5765-516                                            *
*  5765-513                                            *
*  (C) Copyright IBM Corp. 1993, 1997                  *
*                                                      *
*  US Government Users Restricted Rights - Use,        *
*  duplication or disclosure restricted by GSA ADP     *
*  Schedule Contract with IBM Corp.                    *
*****************************************************/
/*****************************************************
 *
 *  Filename:       zmqread.c
 *  Creation Date:  01-Feb-93
 *  Sccs Revision:  "@(#)zmqread.c  IBM Version 1.4 Revised
 *                  11/14/94";
 *  Overview:
 *                  Sends messages to the message queue.
 *
 *  Modification History:
 *  Date        Author  Reason
 *  ~~~~        ~~~~~~  ~~~~~~
 *  01-Feb-93           Originate.
 *  30-Jun-94           Modified to remove Non-MQ Constant
 *                      MQGMO_NONE and
 *                      replaced with MQGMO_NO_WAIT.
 *
 *****************************************************/

#include    <stdio.h>
#include    <string.h>
#include    <time.h>
#include    <stdlib.h>
#include    <sys/timeb.h>
#include    "cmqc.h"

#define     NAME_DELIMETER  '#'
#define     USAGE   "\n\007Usage: %s <QueueManager>#<Queue> <#
                        msgs>\n\n"

static char SccsID[] = "@(#)zmqread.c   IBM Version 1.4 Revised
                          11/14/94";
char    *ProgramName;

int main(int argc, char *argv[])
{
    MQCHAR48    QueueManager="  ",
                queue_manager,
                queue;
    char        *message_data;
    MQHCONN     qm_handle;
    MQHOBJ      que_handle;
    MQLONG      comp_code = MQCC_OK,
                reason = MQRC_NONE,
                selector = MQIA_MAX_MSG_LENGTH;
    MQOD        obj_desc = { MQOD_DEFAULT };
    MQMD        msg_desc = { MQMD_DEFAULT };
    MQGMO       get_options = { MQGMO_DEFAULT };
    long        messages_to_read,
                message_length,
                buffer_length = 0,
                read_index,
                NameLength = 0,
                arg_pos = 0;

    ProgramName = *argv;
    if (argc != 3)
    {
        fprintf(stderr, USAGE, ProgramName);
        exit(1);
    }

/*
* Grab queue manager queue name combination.
*/
    NameLength = 0;
    while ( argv[1][arg_pos]
                && arg_pos < MQ_Q_MGR_NAME_LENGTH
                && argv[1][arg_pos] != NAME_DELIMETER)
    {
        queue_manager[NameLength++] = argv[1][arg_pos++];
    }
    if (arg_pos < MQ_Q_MGR_NAME_LENGTH)
        memset((void *)&queue_manager[NameLength], (int)' ',
                            (size_t)(MQ_Q_MGR_NAME_LENGTH -
                        NameLength));
    strcpy(queue, &argv[1][++arg_pos]);
    if (strlen(queue) < MQ_Q_NAME_LENGTH)
        memset(&queue[strlen(queue)], ' ',
                        MQ_Q_NAME_LENGTH - strlen(queue));

    strncpy(obj_desc.ObjectName, queue, sizeof(MQCHAR48));
    strncpy(obj_desc.ObjectQMgrName, queue_manager,
                        sizeof(MQCHAR48));



/*
* Get number of messages.
*/
    messages_to_read = atoi(argv[2]);

/*
* Access and write messages to queue.
*/
    MQCONN(QueueManager, &qm_handle, &comp_code, &reason);
    if (comp_code)
    {
        printf("Failed to read the application table
                        (Reason=%ld)\n", reason);
        return(1);
    }

    MQOPEN(qm_handle, &obj_desc, MQOO_INPUT_SHARED | MQOO_INQUIRE
                        | MQOO_BROWSE,
                                    &que_handle, &comp_code,
                        &reason);
    if (comp_code)
    {
        printf("Failed to attach to queue. Reason = %ld \n",
                        reason);
        return(1);
    }
    MQINQ(qm_handle, que_handle, 1, &selector, 1, &buffer_length,
                        0,
                NULL, &comp_code, &reason);
    if (comp_code)
    {
/*
* Since we do not have a direct path to the queue, lets
* poll the first message to determine the length of the
* message.  We shall then attempt to read a message size of
* the first message size + 10 bytes for good luck!
*
*  The sequece to follow is:
*      1) Read message with only MQGMO_BROWSE_FIRST.
*          This reads a message, but does not delete it.
*      2) Read message with options = MQGMO_UNLOCK.
*          This option is used to unlock the message that
*          was just read.  The option MQGMO_MSG_UNDER_CURSOR
```

```
*          would have performed a delete of the previously
*          read message.
*      3) Set options back to MQGMO_NO_WAIT.
*          This sets up subsequent reads to both read and
*          delete messages in one call to MQGET.
*/
        get_options.Options = MQGMO_BROWSE_FIRST | MQGMO_LOCK;
        MQGET(qm_handle, que_handle, &msg_desc, &get_options,
                                0, message_data, &message_length,
                                    &comp_code, &reason);
        if (comp_code == MQCC_FAILED)
        {
            printf("%s: Failed to get queue size defaulting to
                        75\n",
                                                        ProgramName);
            buffer_length = 75;
        }
        else
        {
            buffer_length = message_length + 20;
            printf("%s: Message length is %d\n", ProgramName,
                        buffer_length);
        }
        get_options.Options = MQGMO_UNLOCK;
        memcpy(msg_desc.MsgId, MQMI_NONE, sizeof(MQBYTE24));
        memcpy(msg_desc.CorrelId, MQCI_NONE, sizeof(MQBYTE24));
        MQGET(qm_handle, que_handle, &msg_desc, &get_options,
                           buffer_length, message_data,
                        &message_length,
                                    &comp_code, &reason);
        get_options.Options = MQGMO_NO_WAIT;
    }
    message_data = (char *)malloc((size_t)buffer_length);

    get_options.Options |= MQGMO_ACCEPT_TRUNCATED_MSG;

    for (read_index = 0; read_index < messages_to_read;
                        ++read_index)
    {

       memcpy(msg_desc.MsgId, MQMI_NONE, sizeof(MQBYTE24));
        memcpy(msg_desc.CorrelId, MQCI_NONE, sizeof(MQBYTE24));
       MQGET(qm_handle, que_handle, &msg_desc, &get_options,
              buffer_length, message_data, &message_length,
                                    &comp_code, &reason);
        if (comp_code)
        {
            if (comp_code != MQCC_WARNING)
            {
                if (reason == MQRC_NO_MSG_AVAILABLE)
                {
                    printf("%s: No messages left to read\n",
                          ProgramName);
                }
                else
                {
                    printf("%s: Failed to read message #%ld.\n",
                          ProgramName,
                                            read_index);
                }
                printf("Failed to get message. Reason = %ld \n",
                          reason);
                break;
            }
            printf("Warning getting message. Reason = %ld \n",
                          reason);
        }
        printf("==> %-75.75s\n", message_data);
    }

    printf("%s: Read %ld of %ld messages!\n", ProgramName,
                            read_index, messages_to_read);
    free(message_data);
    MQCLOSE(qm_handle, &que_handle, MQCO_NONE, &comp_code,
                          &reason);
    if (comp_code)
    {
        printf("Failed to close queue. (Reason = %ld)\n", reason);
        return(1);
    }

    MQDISC(&qm_handle, &comp_code, &reason);
}
```

zmqwrite.c

```
/****************************************************************
*  Licensed Materials - Property of IBM                         *
*                                                               *
*  5765-519                                                     *
*  5765-521                                                     *
*  63H9503,5697-265                                             *
*  5765-520                                                     *
*  5765-516                                                     *
*  5765-513                                                     *
*  (C) Copyright IBM Corp. 1993, 1997                           *
*                                                               *
*  US Government Users Restricted Rights - Use,                 *
*  duplication or disclosure restricted by GSA ADP              *
*  Schedule Contract with IBM Corp.                             *
****************************************************************/
/****************************************************************
 *
 *  Filename:       zmqwrite.c
 *  Creation Date:  01-Feb-93
 *  Sccs Revistion: "@(#)zmqwrite.c IBM Version 1.3 Revised
                        11/14/94";
 *  Overview:
 *                  Sends messages to the message queue.
 *
 *  Modification History:
 *  Date        Author  Reason
 *  ~~~~        ~~~~~~  ~~~~~~
 *  01-Feb-93           Originate.
 *
 ****************************************************************/

#include    <stdio.h>
#include    <string.h>
#include    <time.h>
#include    <stdlib.h>
#include    <sys/timeb.h>
#include    "cmqc.h"

#define     NAME_DELIMETER  '#'

#define     USAGE   \
"\n\007Usage: %s <QueueManager>#<Queue> <# msgs> <msg length>
                        <message>\n\n"

static char SccsID[] = "@(#)zmqwrite.c  IBM Version 1.3 Revised
                        11/14/94";
char    *ProgramName;

int main(int argc, char *argv[])
{
    MQCHAR48    QueueManager="  ",
                queue_manager,
                queue;
    char        *message_data,
                message_number[10],
                *msg_to_send;
    MQHCONN     qm_handle;
    MQHOBJ      que_handle;
    MQLONG      comp_code = MQCC_OK,
                reason = MQRC_NONE;
    MQOD        obj_desc = { MQOD_DEFAULT };
    MQMD        msg_desc = { MQMD_DEFAULT },
                put_msg_desc;
    MQPMO       put_options = { MQPMO_DEFAULT };
    long        messages_to_send,
                message_length,
                buffer_length,
                message_index,
                buffer_index,
                send_index,
                NameLength = 0,
                arg_pos = 0;


    ProgramName = *argv;
    if (argc != 5)
    {
        fprintf(stderr, USAGE, ProgramName);
        exit(1);
    }

/*
* Grab queue manager queue name combination.
*/
    NameLength = 0;
    while ( argv[1][arg_pos]
                && arg_pos < MQ_Q_MGR_NAME_LENGTH
                && argv[1][arg_pos] != NAME_DELIMETER)
    {
        queue_manager[NameLength++] = argv[1][arg_pos++];
    }
    if (arg_pos < MQ_Q_MGR_NAME_LENGTH)
        memset((void *)&queue_manager[NameLength], (int)' ',
                            (size_t)(MQ_Q_MGR_NAME_LENGTH -
                          NameLength));
    strcpy(queue, &argv[1][++arg_pos]);
    if (strlen(queue) < MQ_Q_NAME_LENGTH)
        memset(&queue[strlen(queue)], ' ',
                         MQ_Q_NAME_LENGTH - strlen(queue));
    strncpy(obj_desc.ObjectName, queue, sizeof(MQCHAR48));
    strncpy(obj_desc.ObjectQMgrName, queue_manager,
                          sizeof(MQCHAR48));



/*
* Get number of messages.
*/
    messages_to_send = atoi(argv[2]);

/*
* Get buffer length.
*/
    if ((buffer_length = atoi(argv[3])) < 8)
    {
        fprintf(stderr, "%s: Buffer length must be at least 8\n",
                            ProgramName);
        exit(1);
    }

/*
* Get message data.
*/
    msg_to_send = argv[4];
    message_length = strlen(msg_to_send);

    message_data = (char *)malloc((size_t)buffer_length);
    memset((void *)message_data, (int)' ', (size_t)buffer_length);

    for (buffer_index = 8, message_index = 0;
                    buffer_index < buffer_length;
                        ++buffer_index,
                           message_index = (++message_index %
                          message_length))
    {
        message_data[buffer_index] = msg_to_send[message_index];
    }


/*
* Access and write messages to queue.
*/
    MQCONN(QueueManager, &qm_handle, &comp_code, &reason);
    if (comp_code)
    {
```

```
        printf("Failed to connect to queue manager. (Reason =
                        %ld)\n", reason);
        return(1);
    }

    MQOPEN(qm_handle, &obj_desc, MQOO_OUTPUT, &que_handle,
                                          &comp_code, &reason);
    if (comp_code)
    {
        printf("Failed to open queue. Reason = %ld\n", reason);
        return(1);
    }

    for (send_index = 0; send_index < messages_to_send;
                        ++send_index)
    {
        sprintf(message_number,"[%.04ld]:  ", send_index+1);
        memcpy(message_data, message_number, 8);

        memcpy(&put_msg_desc, &msg_desc, sizeof(MQMD));
        MQPUT(qm_handle, que_handle, &put_msg_desc, &put_options,
                        buffer_length, message_data, &comp_code,
                        &reason);
        if (comp_code)
        {
           printf("Failed to put message #%ld on queue. (Reason =
                        %ld)\n",
                                  send_index + 1, reason);
            break;
        }
    }

    free(message_data);
    MQCLOSE(qm_handle, &que_handle, MQCO_NONE, &comp_code,
                        &reason);
    if (comp_code)
    {
       printf("Failed to close queue.  (Reason = %ld)\n", reason);
        return(1);
    }

    MQDISC(&qm_handle, &comp_code, &reason);
}
```

cmqc.h

```
/****************************************************
*  Licensed Materials - Property of IBM                *
*                                                      *
*  This Module is "Restricted Materials of IBM"        *
*                                                      *
*  5765-519                                            *
*  5765-521                                            *
*  63H9503,5697-265                                    *
*  5765-520                                            *
*  5765-516                                            *
*  5765-513                                            *
*  (C) Copyright IBM Corp. 1993, 1997                  *
*                                                      *
*  See Copyright instructions.                         *
*                                                      *
*  US Government Users Restricted Rights - Use,        *
*  duplication or disclosure restricted by GSA ADP     *
*  Schedule Contract with IBM Corp.                    *
****************************************************/
/*
 *  Filename:       cmqc.h
 *  Creation Date:  27-Oct-92
 *  Sccs Revision:  "@(#)cmqc.h IBM Version 1.6 Revised 11/14/94"
 *  Overview:
 *       This file contains the defines and typedefs needed
 *       to establish the datatypes used by MQI and MQ-FAP.
 *
 *  Modification History:
 *  Date        Author  Reason
 *  ~~~~        ~~~~~~  ~~~~~~
 *  27-Oct-92           Originate.
 *  30-Oct-92           Added function prototypes for both ANSI
 *                      and non-ANSI compilers.
 *  10-May-92           Changed file name from mqi.h to cmqc.h.
 *  05-Jul-94           Modified check for ANSI to __STDC__
 *  28-Jul-94           Modified the AccountingToken field of
 *                      MQMD_DEFAULT to be 32 NULL's rather than
 *                      32 SPACE's.
 *
 */

#ifndef     CMQC_H
#define     CMQC_H

#include    <mqtypes.h>
#include    <mqconst.h>

#if defined(_MSDOS) || defined(__OS2__)
#pragma pack(1)
#endif

#if defined(_cplusplus) || defined(__cplusplus)
   extern "C" {
#endif

/*
 *  The MQGMO structure is an input variable for passing options
                          to the
 *  MQGET call.
 */
struct tagMQGMO   {
    MQCHAR4   StrucId;          /* Structure identifier 'GMO '  */
    MQLONG    Version;          /* Structure version number     */
    MQLONG    Options;          /* Call options                 */
    MQLONG    WaitInterval;     /* Wait interval                */
    MQLONG    Signal1;          /* Operating-system signal      */
    MQLONG    Signal2;          /* Reserved                     */
    MQCHAR48  ResolvedQName;    /* Resolved queue name         */
};
typedef struct tagMQGMO MQGMO;
typedef MQGMO *PMQGMO;

#define MQGMO_DEFAULT   MQGMO_STRUC_ID_ARRAY, \
                        MQGMO_VERSION_1, \
                        0, \
                        0, \
                        0, \
                        0, \
                        ""


/*
 *  The MQMD structure is used to describe the attributes of a
                          message.
 *  It is an input/output variable for MQGET, MQPUT, and MQPUT1
                          calls.
 */

struct tagMQMD    {
    MQCHAR4   StrucId;          /* Structure identifier 'MD  '  */
    MQLONG    Version;          /* Structure version number     */
    MQLONG    Report;           /* Report options               */
    MQLONG    MsgType;          /* Message type                 */
    MQLONG    Expiry;           /* Expiry time                  */
    MQLONG    Feedback;         /* Exception feedback           */
    MQLONG    Encoding;         /* Data encoding                */
    MQLONG    CodedCharSetId; /*Coded character set identifier*/
    MQCHAR8   Format;           /* Format name                  */
    MQLONG    Priority;         /* Priority                     */
    MQLONG    Persistence;      /* Persistence                  */
    MQBYTE24  MsgId;            /* Message identifier           */
    MQBYTE24  CorrelId;         /* Correlation identifier       */
    MQLONG    BackoutCount;     /* Backout counter for MQGET    */
    MQCHAR48  ReplyToQ;         /* Queue name for replies       */
    MQCHAR48  ReplyToQMgr;       /* QM name for ReplyTo queue */
    MQCHAR12  UserIdentifier;    /* Reserved                     */
    MQBYTE32  AccountingToken;   /* Reserved                     */
    MQCHAR32  ApplIdentityData;  /* Reserved                     */
    MQLONG    PutApplType;       /* Reserved                     */
    MQCHAR28  PutApplName;       /* Reserved                     */
    MQCHAR8   PutDate;           /* Reserved                     */
    MQCHAR8   PutTime;           /* Reserved                     */
    MQCHAR4   ApplOriginData;    /* Reserved                     */
};
typedef struct tagMQMD  MQMD;
typedef MQMD    *PMQMD;

#define MQMD_DEFAULT MQMD_STRUC_ID_ARRAY,\
                     MQMD_VERSION_1,\
                     0,\
                     MQMT_DATAGRAM,\
                     MQEI_UNLIMITED,\
                     MQFB_NONE,\
                     MQENC_NATIVE,\
                     MQCCSI_Q_MGR,\
                     { ' ',' ',' ',' ',' ',' ',' ',' ' }, \
                     0,\
                     MQPER_PERSISTENCE_AS_Q_DEF,\
                     MQMI_NONE_ARRAY,\
                     MQCI_NONE_ARRAY,\
                     0,\
                     "",\
                     "",\
                     "",\
                     { '\0','\0','\0','\0','\0','\0','\0','\0', \
                       '\0','\0','\0','\0','\0','\0','\0','\0', \
                       '\0','\0','\0','\0','\0','\0','\0','\0', \
                      '\0','\0','\0','\0','\0','\0','\0','\0' }, \
                     "",\
                     0,\
                     "",\
                     { ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ' }, \
                     { ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ' }, \
                     ""
```

```
/*
 *  The MQOD structure is used to specify a queue object.
 */
struct tagMQOD    {
    MQCHAR4   StrucId;          /* Structure identifier 'OD  '  */
    MQLONG    Version;          /* Structure version number     */
    MQLONG    ObjectType;       /* Object type                  */
    MQCHAR48  ObjectName;       /* Object name                  */
    MQCHAR48  ObjectQMgrName;   /* Object queue manager name    */
    MQCHAR48  DynamicQName;     /* Dynamic queue name           */
    MQCHAR12  AlternateUserId;  /* Reserved                     */
};
typedef struct tagMQOD  MQOD;
typedef MQOD *PMQOD;

#define MQOD_DEFAULT MQOD_STRUC_ID_ARRAY,\
                     MQOD_VERSION_1,\
                     MQOT_Q,\
                     "",\
                     "",\
                     "*"


/*
 *  The MQPMO structure is an input variable for passing options
                        to the
 *  MQPUT and MQPUT1 calls.
 */
struct tagMQPMO   {
    MQCHAR4   StrucId;          /* Structure identifier 'PMO '  */
    MQLONG    Version;          /* Structure version number     */
    MQLONG    Options;          /* Call options                 */
    MQLONG    Timeout;          /* Reserved; must be -1         */
    MQHOBJ    Context;          /* Reserved                     */
    MQLONG    KnownDestCount;   /* Reserved                     */
    MQLONG    UnknownDestCount; /* Reserved                     */
    MQLONG    InvalidDestCount; /* Reserved                     */
    MQCHAR48  ResolvedQName;    /* Resolved queue name          */
    MQCHAR48  ResolvedQMgrName; /* Resolved queue manager name  */
};
typedef struct tagMQPMO MQPMO;
typedef MQPMO *PMQPMO;

#define MQPMO_DEFAULT MQPMO_STRUC_ID_ARRAY,\
                      MQPMO_VERSION_1,\
                      0,\
                      -1,\
                      0,\
                      0,\
                      0,\
                      0,\
                      "",\
                      ""

/*****************************************************************/
/*  MQTM Structure                                               */
/*****************************************************************/

struct tagMQTM {
  MQCHAR4    StrucId;      /* Structure identifier */
  MQLONG     Version;      /* Structure version number */
  MQCHAR48   QName;        /* Name of triggered queue */
  MQCHAR48   ProcessName;  /* Name of process object */
  MQCHAR64   TriggerData;  /* Trigger data */
  MQLONG     ApplType;     /* Application type */
  MQCHAR256  ApplId;       /* Application identifier */
  MQCHAR128  EnvData;      /* Environment data */
  MQCHAR128  UserData;     /* User data */
 };
typedef struct tagMQTM  MQTM;
typedef MQTM *PMQTM;

#define MQTM_DEFAULT MQTM_STRUC_ID_ARRAY,\
                     MQTM_VERSION_1,\
                     "",\
                     "",\
                     "",\
                     0,\
                     "",\
                     "",\
                     ""

/*****************************************************************/
/*  MQTM Structure                                               */
/*****************************************************************/
struct tagMQDLH {
    MQCHAR4     StrucId;
    MQLONG      Version;
    MQLONG      Reason;
    MQCHAR48    DestQName;
    MQCHAR48    DestQMgrName;
    MQLONG      Encoding;
    MQLONG      CodedCharSetId;
    MQCHAR8     Format;
    MQLONG      PutApplType;
    MQCHAR28    PutApplName;
    MQCHAR8     PutDate;
    MQCHAR8     PutTime;
};
typedef struct tagMQDLH MQDLH;

#define MQDLH_DEFAULT   MQDLH_STRUC_ID_ARRAY\
                        MQDLH_VERSION_1 \
                        MQRC_NONE, \

                          "
                                    ", \

                          "
                                    ", \
                        0, \
                        0, \
                        "       ", \
                        MQAT_NO_CONTEXT, \
                        "                               ", \
                        "       ", \
                        "       "


/*
 *  This section contains both ansi and non-ansi prototypes
 *  for the MQI functions.
 */

#if defined(__STDC__) || defined(_MSDOS) || defined(__OS2__)
void MQCONN( MQCHAR48    Name,  /* Name of queue manager (input)*/
            PMQHCONN    Hconn,     /* Connection handle (output)*/
             PMQLONG     CompCode,   /* Completion code (output)*/
             PMQLONG     Reason  /* Code qualifying CompCode
                                    (output)*/
            );

void MQDISC( PMQHCONN    Hconn, /* Connection handle.
                                   (input/output)*/
              PMQLONG     CompCode,   /* Completion code (output)*/
             PMQLONG    Reason  /* Code qualifying CompCode
                                   (output)*/
            );

void MQOPEN( MQHCONN     Hconn,   /* Connection handle. (input)*/
             PMQOD      ObjDesc, /* Object descriptor.
                                   (input/output)*/
            MQLONG     Options, /* Action control options (input)*/
             PMQHOBJ    Hobj,          /* Object handle (output)*/
             PMQLONG    CompCode,  /* Completion code (output)*/
            PMQLONG   Reason   /* Code qualifying CompCode
                                   (output)*/
             );

void MQCLOSE(MQHCONN Hconn, /*Connection handle. (input)*/
```

```
            PMQHOBJ Hobj, /*Object handle (input/output)*/
             MQLONG Options, /*Action control options (input)*/
            PMQLONG CompCode, /*Completion code (output)*/
             PMQLONG Reason /*Code qualifying CompCode (output)*/
             );

void MQGET(  MQHCONN Hconn, /* Connection handle. (input) */
              MQHOBJ  Hobj, /* Object handle (input) */
              PMQMD   MsgDesc, /* Message descriptor
                           (input/output)*/
             PMQGMO  GetMsgOpts, /*action control (input/output)*/
             MQLONG      BufferLength,  /* Buffer length (input) */
              PMQVOID     Buffer, /* Retrieved message (output) */
             PMQLONG    DataLength, /*Length of message (output)*/
              PMQLONG     CompCode, /* Completion code (output) */
              PMQLONG     Reason /* Code qualifying CompCode
                           (output)*/
             );

void MQPUT( MQHCONN  Hconn, /* Connection handle. (input) */
             MQHOBJ   Hobj,  /* Object handle (output) */
            PMQMD   MsgDesc, /* Message descriptor (input/output)*/
             PMQPMO  PutMsgOpts, /*action control (input/output)*/
            MQLONG  BufferLength,  /* Buffer length (input)*/
            PMQVOID  Buffer,/* Message data (input)*/
            PMQLONG  CompCode, /* Completion code (output)*/
             PMQLONG  Reason /* Code qualifying CompCode (output)*/
             );

void MQPUT1( MQHCONN  Hconn, /* Connection handle. (input) */
              PMQOD  ObjDesc, /* Object descriptor (input) */
             PMQMD  MsgDesc, /* Message descriptor (input/output)*/
             PMQPMO PutMsgOpts, /* action control (input/output) */
             MQLONG BufferLength, /* Buffer length (input) */
             PMQVOID Buffer,  /* Message data (input) */
              PMQLONG  CompCode, /* Completion code (output) */
              PMQLONG Reason /* Code qualifying CompCode (output)*/
             );

void MQINQ(  MQHCONN  Hconn, /* Connection handle. (input) */
           MQHOBJ   Hobj, /* Object handle (input) */
            MQLONG  SelectorCount, /* Count of selectors (input)*/
           PMQLONG  Selectors, /*attribute selectors (input)*/
           MQLONG IntAttrCount,/*Count of integer attribs(input)*/
           PMQLONG IntAttrs, /* Array of integer attribs (output)*/
            MQLONG CharAttrLength,/* Length of char attrib buf
                         (input)*/
            PMQCHAR CharAttrs, /* Character attributes. (output)*/
           PMQLONG CompCode, /* Completion code (output) */
            PMQLONG Reason /* Code qualifying CompCode (output)*/
             );


#else   /*  #ifdef __STDC__  */

void MQCONN();
void MQDISC();
void MQOPEN();
void MQCLOSE();
void MQGET();
void MQPUT();
void MQPUT1();
void MQINQ();

#endif  /*  #ifdef __STDC__  */

#if defined(_cplusplus) || defined(__cplusplus)
   }
#endif
#endif      /* #ifndef CMQC_H  */
```

mqconst.h

```
/****************************************************
*  Licensed Materials - Property of IBM             *
*                                                   *
*  This Module is "Restricted Materials of IBM"     *
*                                                   *
*  5765-519                                         *
*  5765-521                                         *
*  63H9503,5697-265                                 *
*  5765-520                                         *
*  5765-516                                         *
*  5765-513                                         *
*  (C) Copyright IBM Corp. 1993, 1997               *
*                                                   *
*  See Copyright instructions.                      *
*                                                   *
*  US Government Users Restricted Rights - Use,     *
*  duplication or disclosure restricted by GSA ADP  *
*  Schedule Contract with IBM Corp.                 *
****************************************************/
/*
 *  Filename:       mqconst.h
 *  Creation Date:  26-Oct-92
 *  Sccs Revision: "@(#)mqconst.h IBM Version 1.10 Revised
                        11/14/94";
 *  Overview:
 *    This header file contains all the define statements
 *    that were defined in the original IBM specifications
 *    document "ssisroot.lis".
 *
 *  Modification History:
 *  Date        Author  Reason
 *  ~~~~        ~~~~~~  ~~~~~~
 *  26-Oct-92    Originate
 *  18-Aug-93    Added new reason code MQRC_NO_MSG_LOCKED, and
 *               changed MQGMO_BROWSE_MSG_UNDER_CURSOR to be
 *               MQGMO_UNLOCK.
 *  12-Jul-94    Modified all instances of _INIT to be _ARRAY.
                        This
 *              conforms to IBM's own internal naming conventions.
 *               There are corresponding changes in cmqc.h.  Also
                        added
 *               another null to the MQMI_NONE and MQCI_NONE
                        definitions.
 *              This will circumvent the compiler problem reported
 *               with AIX.  Definitions of MQMI_NONE_ARRAY and
 *               MQCI_NONE_ARRAY have also been made for use in
 *               cmqc.h.
 *  28-Jul-94    Added definitions of MQTM_STRUC_ID_ARRAY,
 *               MQTM_STRUC_ID, and MQTM_VERSION_1.
 *  02-Aug-94    Added definition of MQFB_DUPLICATE_MSG_SENT(274).
 *  20-Sep-94    Added definition of MQRC_DATABASE_DEADLOCK(2231)
 *  18-Oct-94    Changed name of MQRC_DATABASE_DEADLOCK to be
 *               MQRC_DATABASE_ERROR.  In this way the error code
                        is
 *              more flexible.  The end user must now examine the
 *               proper global variables supplied by the database
 *               in use for further explanation of the error.
 *
 */

#ifndef MQCONST_H
#define MQCONST_H

#if defined(_MSDOS) || defined(__OS2__)
#pragma pack(1)
#endif

#if defined(_cplusplus) || defined(__cplusplus)
   extern "C" {
#endif

/*
 *  Lengths of character string and byte fields
 */

#define MQ_CREATION_DATE_LENGTH               12
#define MQ_CREATION_TIME_LENGTH                8
#define MQ_PROCESS_APPL_ID_LENGTH            256
#define MQ_PROCESS_DESC_LENGTH                64
#define MQ_PROCESS_ENV_DATA_LENGTH           128
#define MQ_PROCESS_NAME_LENGTH                48
#define MQ_PROCESS_USER_DATA_LENGTH          128
#define MQ_Q_DESC_LENGTH                      64
#define MQ_Q_NAME_LENGTH                      48
#define MQ_Q_MGR_DESC_LENGTH                  64
#define MQ_Q_MGR_NAME_LENGTH                  48
#define MQ_TRIGGER_DATA_LENGTH                64

/*
 *  Application type
 */

#define MQAT_USER_FIRST                    65536L
#define MQAT_USER_LAST                   999999999L
#define MQAT_NO_CONTEXT                        0L
#define MQAT_OS2                               4L
#define MQAT_DOS                               5L
#define MQAT_AIX                               6L
#define MQAT_QMGR                              7L
#define MQAT_OS400                             8L
#define MQAT_WINDOWS                           9L
#define MQAT_CICS_VSE                         10L
#define MQAT_VMS                              12L
#define MQAT_GUARDIAN                         13L
#define MQAT_VOS                              14L

/*
 *  Character attribute selectors
 */

#define MQCA_Q_NAME                         2016L
#define MQCA_Q_DESC                         2013L
#define MQCA_PROCESS_NAME                   2012L
#define MQCA_INITIATION_Q_NAME              2008L
#define MQCA_CREATION_DATE                  2004L
#define MQCA_CREATION_TIME                  2005L
#define MQCA_REMOTE_Q_NAME                  2018L
#define MQCA_REMOTE_Q_MGR_NAME              2017L
#define MQCA_BASE_Q_NAME                    2002L
#define MQCA_XMIT_Q_NAME                    2024L

/*
 *  Coded character set identifier
 */

#define MQCCSI_Q_MGR                           0L

/*
 *  Completion codes
 */

#define MQCC_OK                                0L
#define MQCC_WARNING                           1L
#define MQCC_FAILED                            2L

/*
 *  Correlation identifier
 */

/*
 *  Note that in this definition there is one less
 *  NULL character.  This is because the definition
 *  as string automatically appends a NULL
 *  character to the end.
 */
```

```
#define MQCI_NONE
                        "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
                        \0\0\0\0\0"
#define MQCI_NONE_ARRAY {
                        '\0','\0','\0','\0','\0','\0','\0','\0'
                        , \
                       '\0','\0','\0','\0','\0','\0','\0','\0', \
                       '\0','\0','\0','\0','\0','\0','\0','\0' }

/*
 *  CPI level
 */

#define MQCL_LEVEL_1                    100L

/*
 *  Close options
 */
#define MQCO_NONE                         0L

/*
 *  Dead letter queue header structure identifier.
 */
#define MQDLH_STRUC_ID_ARRAY {'D','L','H',' '}
#define MQDLH_STRUC_ID                "DLH "

/*
 *  Dead letter queue header version.
 */
#define MQDLH_VERSION_1                   1L

/*
 *  Encoding masks
 */

#define MQENC_INTEGER_MASK          0x0000000F
#define MQENC_DECIMAL_MASK          0x000000F0
#define MQENC_FLOAT_MASK            0x00000F00
#define MQENC_RESERVED_MASK         0xFFFFF000

/*
 *  Encoding for binary integers
 */
#define MQENC_INTEGER_UNDEFINED           0L
#define MQENC_INTEGER_NORMAL              1L
#define MQENC_INTEGER_REVERSED            2L

/*
 *  Encoding for packed-decimal integers
 */
#define MQENC_DECIMAL_UNDEFINED           0L
#define MQENC_DECIMAL_NORMAL             16L
#define MQENC_DECIMAL_REVERSED           32L

/*
 *  Encoding for floating-point numbers
 */
#define MQENC_FLOAT_UNDEFINED             0L
#define MQENC_FLOAT_IEEE_NORMAL         256L
#define MQENC_FLOAT_IEEE_REVERSED       512L
#define MQENC_FLOAT_S390                768L

/*
 *  Encoding
 */
#if defined(_AIX) || defined(sparc) || defined(__hpux)
#define MQENC_NATIVE (MQENC_INTEGER_NORMAL | MQENC_DECIMAL_NORMAL
                      | MQENC_FLOAT_IEEE_NORMAL)
#else   /*  _AIX    */

#if defined(__OS2__) || defined(_MSDOS) || defined(INTEL) ||
                       defined(i386)
#define MQENC_NATIVE (MQENC_INTEGER_REVERSED |
                      MQENC_DECIMAL_REVERSED |
                      MQENC_FLOAT_IEEE_REVERSED)
#endif  /*  __OS2__ || _MSDOS   */

#endif  /*  _AIX    */

/* Expiry Value */
#define MQEI_UNLIMITED                   -1

/*
 *  Feedback
 */

#define MQFB_NONE                         0L
#define MQFB_QUIT                       256L
#define MQFB_SYSTEM_FIRST                 1L
#define MQFB_SYSTEM_LAST                65535L
#define MQFB_APPL_FIRST                 65536L
#define MQFB_APPL_LAST                999999999L

#define MQFB_DUPLICATE_MSG_SEQ_NUMBER     272L
#define MQFB_ALSO_PUT_ON_REMOTE_DEAD_Q    273L
#define MQFB_DUPLICATE_MSG_SENT           274L

/*
 *  Format
 */
#define MQFMT_NONE                    "        "
#define MQFMT_DEAD_LETTER_HEADER      "MQDEAD  "
#define MQFMT_TRIGGER                 "MQTRIG  "
#define MQFMT_XMIT_Q_HEADER           "MQXMIT  "

/*
 *  Get message options
 */

#define MQGMO_WAIT                        1L
#define MQGMO_NO_WAIT                     0L
#define MQGMO_SYNCPOINT                   2L
#define MQGMO_NO_SYNCPOINT                4L
#define MQGMO_BROWSE_FIRST               16L
#define MQGMO_BROWSE_NEXT                32L
#define MQGMO_ACCEPT_TRUNCATED_MSG       64L
#define MQGMO_SET_SIGNAL                  8L
#define MQGMO_MSG_UNDER_CURSOR          256L
#define MQGMO_LOCK                      512L
#define MQGMO_UNLOCK                   1024L

/* Wait Interval */
#define MQWI_UNLIMITED -1L

/*
 *  Get message options structure identifier
 */

#define MQGMO_STRUC_ID_ARRAY    {'G','M','O',' '}
#define MQGMO_STRUC_ID                "GMO "

/*
 *  Get message options version
 */

#define MQGMO_VERSION_1                   1L

/*
 *  Integer attribute selectors
 */

#define MQIA_FIRST                        1L
#define MQIA_USAGE                       12L
#define MQIA_Q_TYPE                      20L
#define MQIA_INHIBIT_GET                  9L
#define MQIA_INHIBIT_PUT                 10L
#define MQIA_DEF_PERSISTENCE              5L
#define MQIA_MAX_Q_DEPTH                 15L
```

```
#define MQIA_MAX_MSG_LENGTH                   13
#define MQIA_SHAREABILITY                    23L
#define MQIA_DEFINITION_TYPE                  7L
#define MQIA_TRIGGER_CONTROL                 24L
#define MQIA_TRIGGER_TYPE                    28L
#define MQIA_OPEN_INPUT_COUNT                17L
#define MQIA_OPEN_OUTPUT_COUNT               18L
#define MQIA_CURRENT_Q_DEPTH                  3L
#define MQIA_LAST                          2000L

/*
 *  Integer attribute value
 */

#define MQIAV_NOT_APPLICABLE               -1L

/*
 *  Message descriptor structure identifier
 */
#define MQMD_STRUC_ID_ARRAY  {'M','D',' ',' '}
#define MQMD_STRUC_ID                    "MD  "

/*
 *  Message descriptor version
 */

#define MQMD_VERSION_1                        1L

/*
 *  Message identifier
 */

#define MQMI_NONE
                         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
                         \0\0\0\0\0\0"
#define MQMI_NONE_ARRAY {
                         '\0','\0','\0','\0','\0','\0','\0','\0'
                         , \
   '\0','\0','\0','\0','\0','\0','\0','\0', \
    '\0','\0','\0','\0','\0','\0','\0','\0' }

/*
 *  Message type
 */
#define MQMT_REQUEST                          1L
#define MQMT_REPLY                            2L
#define MQMT_DATAGRAM                         8L
#define MQMT_REPORT                           4L

/* Report Options */
#define MQRO_NONE                   0x00000000L

/*
 *  Object descriptor structure identifier
 */

#define MQOD_STRUC_ID_ARRAY  {'O','D',' ',' '}
#define MQOD_STRUC_ID                    "OD  "

/*
 *  Object descriptor version
 */

#define MQOD_VERSION_1                        1L

/*
 *  Open options
 */

#define MQOO_INPUT_SHARED                     2L
#define MQOO_INPUT_EXCLUSIVE                  4L
#define MQOO_BROWSE                           8L
#define MQOO_OUTPUT                          16L
#define MQOO_INQUIRE                         32L

/*
 *  Object type
 */

#define MQOT_Q                                1L

/*
 *  Persistence
 */

#define MQPER_PERSISTENT                      1L
#define MQPER_PERSISTENCE_AS_Q_DEF            2L

/*
 *  Put message options structure identifier
 */

#define MQPMO_STRUC_ID_ARRAY    {'P','M','O',' '}
#define MQPMO_STRUC_ID                     "PMO "

/*
 *  Put Message options version
 */

#define MQPMO_VERSION_1                          1L

#define MQPMO_SYNCPOINT                          2L
#define MQPMO_NO_SYNCPOINT                       4L

/*
 *  MQTM definitions.
 */
#define MQTM_STRUC_ID_ARRAY     {'T','M',' ',' '}
#define MQTM_STRUC_ID                      "TM  "
#define MQTM_VERSION_1                           1L

/*
 *  Inquire object attributes.
 */
#define MQIQ_MAX_SELECTORS                    256L

/*
 *  Inhibit get
 */
#define MQQA_GET_INHIBITED                       1L
#define MQQA_GET_ALLOWED                         0L

/*
 *  Inhibit put
 */
#define MQQA_PUT_INHIBITED                       1L
#define MQQA_PUT_ALLOWED                         0L

/*
 *  Queue shareability
 */

#define MQQA_SHAREABLE                           1L
#define MQQA_NOT_SHAREABLE                       0L

/*
 *  Queue definition type
 */

#define MQQDT_PREDEFINED                         1L

/*
 *  Queue type
 */

#define MQQT_LOCAL                               1L
#define MQQT_ALIAS                               3L
```

```
#define MQQT_REMOTE                             6L

/*
 *  Reason codes
 */

#define MQRC_ACCESS_RESTRICTED              2000L
#define MQRC_ALIAS_BASE_Q_TYPE_ERROR        2001L
#define MQRC_ALREADY_CONNECTED              2002L
#define MQRC_BUFFER_ERROR                   2004L
#define MQRC_BUFFER_LENGTH_ERROR            2005L
#define MQRC_CHAR_ATTR_LENGTH_ERROR         2006L
#define MQRC_CHAR_ATTRS_ERROR               2007L
#define MQRC_CHAR_ATTRS_TOO_SHORT           2008L
#define MQRC_CONNECTION_BROKEN              2009L
#define MQRC_DATA_LENGTH_ERROR              2010L
#define MQRC_EXPIRY_ERROR                   2013L
#define MQRC_FEEDBACK_ERROR                 2014L
#define MQRC_GET_INHIBITED                  2016L
#define MQRC_HANDLE_NOT_AVAILABLE           2017L
#define MQRC_HCONN_ERROR                    2018L
#define MQRC_HOBJ_ERROR                     2019L
#define MQRC_INT_ATTR_COUNT_ERROR           2021L
#define MQRC_INT_ATTR_COUNT_TOO_SMALL       2022L
#define MQRC_INT_ATTRS_ARRAY_ERROR          2023L
#define MQRC_LOCK_NOT_AVAILABLE             2024L
#define MQRC_MAX_CONNS_LIMIT_REACHED        2025L
#define MQRC_MD_ERROR                       2026L
#define MQRC_MISSING_REPLY_TO_Q             2027L
#define MQRC_MSG_TYPE_ERROR                 2029L
#define MQRC_MSG_TOO_BIG_FOR_Q              2030L
#define MQRC_MSG_TOO_BIG_FOR_Q_MGR          2031L
#define MQRC_NO_MSG_AVAILABLE               2033L
#define MQRC_NO_MSG_UNDER_CURSOR            2034L
#define MQRC_NONE                              0L
#define MQRC_NOT_AUTHORIZED                 2035L
#define MQRC_NOT_OPEN_FOR_BROWSE            2036L
#define MQRC_NOT_OPEN_FOR_INPUT             2037L
#define MQRC_NOT_OPEN_FOR_INQUIRE           2038L
#define MQRC_NOT_OPEN_FOR_OUTPUT            2039L
#define MQRC_OBJECT_CHANGED                 2041L
#define MQRC_OBJECT_IN_USE                  2042L
#define MQRC_OBJECT_TYPE_ERROR              2043L
#define MQRC_OD_ERROR                       2044L
#define MQRC_OPTION_NOT_VALID_FOR_TYPE      2045L
#define MQRC_OPTIONS_ERROR                  2046L
#define MQRC_PERSISTENCE_ERROR              2047L
#define MQRC_PRIORITY_EXCEEDS_MAXIMUM       2049L
#define MQRC_PRIORITY_ERROR                 2050L
#define MQRC_PUT_INHIBITED                  2051L
#define MQRC_Q_FULL                         2053L
#define MQRC_Q_SPACE_NOT_AVAILABLE          2056L
#define MQRC_Q_MGR_NAME_ERROR               2058L
#define MQRC_Q_MGR_NOT_AVAILABLE            2059L
#define MQRC_REPORT_OPTIONS_ERROR           2061L
#define MQRC_SECURITY_ERROR                 2063L
#define MQRC_SELECTOR_COUNT_ERROR           2065L
#define MQRC_SELECTOR_LIMIT_EXCEEDED        2066L
#define MQRC_SELECTOR_ERROR                 2067L
#define MQRC_SELECTOR_NOT_FOR_TYPE          2068L
#define MQRC_SIGNAL_OUTSTANDING             2069L
#define MQRC_SIGNAL_REQUEST_ACCEPTED        2070L
#define MQRC_STORAGE_NOT_AVAILABLE          2071L
#define MQRC_SYNCPOINT_NOT_AVAILABLE        2072L
#define MQRC_TRUNCATED_MSG_ACCEPTED         2079L
#define MQRC_TRUNCATED_MSG_FAILED           2080L
#define MQRC_UNEXPECTED_CONNECT_ERROR       2081L
#define MQRC_UNKNOWN_ALIAS_BASE_Q           2082L
#define MQRC_UNKNOWN_OBJECT_NAME            2085L
#define MQRC_UNKNOWN_OBJECT_Q_MGR           2086L
#define MQRC_UNKNOWN_REMOTE_Q_MGR           2087L
#define MQRC_WAIT_INTERVAL_ERROR            2090L
#define MQRC_XMIT_Q_TYPE_ERROR              2091L
#define MQRC_XMIT_Q_USAGE_ERROR             2092L
#define MQRC_SIGNAL1_ERROR                  2099L
#define MQRC_PMO_ERROR                      2173L
#define MQRC_GMO_ERROR                      2186L
#define MQRC_UNEXPECTED_ERROR               2195L
#define MQRC_UNKNOWN_XMIT_Q                 2196L
#define MQRC_FILE_SYSTEM_ERROR              2208L
#define MQRC_MSG_ID_ERROR                   2206L
#define MQRC_CORREL_ID_ERROR                2207L
#define MQRC_NO_MSG_LOCKED                  2209L
#define MQRC_DATABASE_ERROR                 2231L
#define MQRC_INDEX_LOCKED                   7777L

/*
 *  Syncpoint
 */
#define MQSP_AVAILABLE                         1L

/*
 *  Trigger controls
 */

#define MQTC_OFF                               0L
#define MQTC_ON                                1L

/*
 *  Trigger type
 */
#define MQTT_NONE                              0L
#define MQTT_FIRST                             1L
#define MQTT_EVERY                             2L

/*
 *  Usage
 */
#define MQUS_NORMAL                            0L
#define MQUS_TRANSMISSION                      1L

#if defined(_cplusplus) || defined(__cplusplus)
   }
#endif
#endif      /* #ifdef MQCONST_H */
```

mqtypes.h

```
/****************************************************
*  Licensed Materials - Property of IBM             *
*                                                   *
*  This Module is "Restricted Materials of IBM"     *
*                                                   *
*  5765-519                                         *
*  5765-521                                         *
*  63H9503,5697-265                                 *
*  5765-520                                         *
*  5765-516                                         *
*  5765-513                                         *
*  (C) Copyright IBM Corp. 1993, 1997               *
*                                                   *
*  See Copyright instructions.                      *
*                                                   *
*  US Government Users Restricted Rights - Use,     *
*  duplication or disclosure restricted by GSA ADP  *
*  Schedule Contract with IBM Corp.                 *
****************************************************/
/*
 *  Filename:       mqtypes.h
 *  Creation Date:  26-Oct-92
 *  Sccs Revision:  "@(#)mqtypes.h IBM Version 1.2 Revised
                        11/14/94"
 *  Overview:
 *    This file contains the defines and typedefs needed
 *    to establish the datatypes used by MQI and MQ-FAP.
 *
 *  Modification History:
 *  Date        Author  Reason
 *  ~~~~        ~~~~~~  ~~~~~~
 *  26-Oct-92           Originate.
 *
 */

#ifndef MQTYPES_H
#define MQTYPES_H

#if defined(_MSDOS) || defined(__OS2__)
#pragma pack(1)
#endif

#if defined(_cplusplus) || defined(__cplusplus)
   extern "C" {
#endif

typedef unsigned char   MQBYTE;
typedef MQBYTE *        PMQBYTE;
typedef MQBYTE          MQBYTE24[24];
typedef MQBYTE          MQBYTE8[8];
typedef MQBYTE8 *       PMQBYTE8;
typedef MQBYTE24 *      PMQBYTE24;
typedef MQBYTE          MQBYTE32[32];
typedef MQBYTE32 *      PMQBYTE32;

typedef char            MQCHAR;
typedef MQCHAR *        PMQCHAR;
typedef MQCHAR          MQCHAR4[4];
typedef MQCHAR4 *       PMQCHAR4;
typedef MQCHAR          MQCHAR8[8];
typedef MQCHAR8 *       PMQCHAR8;
typedef MQCHAR          MQCHAR12[12];
typedef MQCHAR12 *      PMQCHAR12;
typedef MQCHAR          MQCHAR24[24];
typedef MQCHAR24 *      PMQCHAR24;
typedef MQCHAR          MQCHAR28[28];
typedef MQCHAR28 *      PMQCHAR28;
typedef MQCHAR          MQCHAR32[32];
typedef MQCHAR32 *      PMQCHAR32;
typedef MQCHAR          MQCHAR48[48];
typedef MQCHAR48 *      PMQCHAR48;
typedef MQCHAR          MQCHAR64[64];
typedef MQCHAR64 *      PMQCHAR64;
typedef MQCHAR          MQCHAR128[128];
typedef MQCHAR128 *     PMQCHAR128;
typedef MQCHAR          MQCHAR256[256];
typedef MQCHAR256 *     PMQCHAR256;

typedef long            MQLONG;
typedef MQLONG *        PMQLONG;

typedef MQLONG          MQHCONN;
typedef MQHCONN *       PMQHCONN;

typedef MQLONG          MQHOBJ;
typedef MQHOBJ *        PMQHOBJ;

typedef void            MQVOID;
typedef MQVOID *        PMQVOID;
typedef PMQVOID *       PPMQVOID;

#if defined(_cplusplus) || defined(__cplusplus)
   }
#endif
#endif      /* #ifndef  MQTYPES_H   */
```

Appendix C. C programming language examples

This appendix provides examples of how to invoke message-queuing calls. It also shows the various parameters and data types declared in C Programming languages. The information in this appendix is provided to help understand the C application programs that use message queuing.

This section contains:

- Language Considerations
- Functions
- Elementary data types
- Structure data types

Language considerations

Header file

A header file is provided as part of the definition of the message queue interface. The header file is summarized in Table 22.

Table 22. Header file

Filename	Contents
CMQC	Function prototypes, data types, named constants

Functions

Parameters that are *input-only* and of the type `MQHCONN`, `MQHOBJ`, or `MQLONG` are passed by value; for all other parameters, the address of the parameter is passed by value.

Not all parameters that are passed by address need to be specified every time a function is invoked. Where a particular parameter is not required, a null pointer can be specified as the parameter on the function invocation in place of the address of the parameter data. Parameters for which this is possible are identified in the call descriptions. See Chapter 6, “Application programming interface” on page 107.

No parameter is returned as the value of the function. In C terminology, this means that all functions return `void`.

Parameters with undefined data type

The `MQGET`, `MQPUT`, and `MQPUT1` functions each have one parameter that has an undefined data type, namely the *Buffer* parameter. This parameter is used to send and receive the application's message data.

Parameters of this sort are shown in the C examples as arrays of `MQBYTE`. It is valid to declare the parameters in this way, but it more convenient to declare them as the particular structure which describes the layout of the data in the message. The actual function parameter is defined as a pointer-to-void. This means that the address of any sort of data can be specified as the parameter on the function invocation.

Data types

All data types are defined by means of the C `typedef` statement. For each data type, the corresponding pointer data type is also defined. The name of the pointer data type is the elementary name or structure data type prefixed with the letter "P" to denote a pointer.

```
typedef MQLONG *PMQLONG /* pointer to MQLONG */
typedef MQMD *PMQMD; /*pointer to MQMD */
```

Manipulating binary strings

Strings of binary data are declared as one of the `MQBYTEn` data types. Whenever fields of this type are copied, compared, or set, the C functions `memcpy`, `memcmp`, or `memset` should be used. For example:

```
#include <string.h>
#include "CMQC.H"

MQMD MyMsgDesc;

memcpy(MyMsgDesc.MsgId, /* set "MsgId" field to nulls */
    MQMI_NONE, /* ...using named constant */
    sizeof(MyMsgDesc.MsgId));

memset(MyMsgDesc.CorrelId, /* set "CorrelId" field to nulls */
    0x00, /* ...using different method */
    sizeof(MQBYTE24));
```

Do not use the string functions `strcpy`, `strcmp`, `strncpy`, or `strncmp`, because these do not work correctly for data declared with the `MQBYTEn` data types.

Manipulating character strings

When the queue manager returns character data to the application, the queue manager always pads the character data with blanks to the defined length of the field; the queue manager *does not* return null-terminated strings. Therefore, when copying, comparing or concatenating such strings, the string functions `strncpy`, `strncmp`, or `strncat` should be used.

String functions, which require the string to be terminated by a null (`strcpy`, `strcmp`, or `strcat`) were not used. In addition the `strlen` function, used to determine the length of the string was not used. Instead the `sizeof` function to determine the length of the field was used.

Initial values for structures

The header file CMQC defines various macro variables that may be used to provide initial values for the message queuing structures when instances of those structures are declared. These macro variables have names of the form "MQXXX_DEFAULT", (where "MQXXX" represents the name of the structure). They are used in the following way:

```
MQMD MyMsgDesc = {MQMD_DEFAULT};
MQPMO MyPutOpts = {MQPMO_DEFAULT};
```

Notational conventions

The sections that follow show how the:

- Functions should be invoked
- Parameters should be declared
- Various data types should be declared

In a number of cases, parameters are arrays whose size is not fixed. For these, a lowercase "n" is used to represent a numeric constant. When the declaration for that parameter is coded, the "n" must be replaced by the numeric value required.

Functions

MQCLOSE

```
MQCLOSE (Hconn, &Hobj, Options, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQHOBJ     Hobj;               /* Object handle */
MQLONG     Options;            /* Options that control the action of MQCLOSE */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQCONN

```
MQCONN (Name, &Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQCHAR48   Name;               /* Name of queue manager */
MQHCONN    Hconn;              /* Connection handle */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQDISC

```
MQDISC (&Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQGET

```
MQGET (Hconn, Hobj, &MsgDesc, &GetMsgOpts, BufferLength, Buffer, &DataLength,
&CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQHOBJ     Hobj;               /* Object handle */
MQMD       MsgDesc;            /* Message descriptor */
MQGMO      GetMsgOpts;         /* Options that control the action of MQGET */
MQLONG     BufferLength;       /* Length in bytes of the buffer area */
MQBYTE     Buffer[n];          /* Area to contain the message data */
MQLONG     DataLength;         /* Length of the message */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQINQ

```
MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
CharAttrLength, CharAttrs, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQHOBJ     Hobj;               /* Object handle */
MQLONG     SelectorCount;      /* Count of selectors */
MQLONG     Selectors[n];       /* Array of attribute selectors */
MQLONG     IntAttrCount;       /* Count of integer attributes */
MQLONG     IntAttrs[n];        /* Array of integer attributes */
MQLONG     CharAttrLength;     /* Length of character attributes buffer */
MQCHAR     CharAttrs[n];       /* Character attributes */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQOPEN

```
MQOPEN (Hconn, &ObjDesc, Options, &Hobj, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQOD       ObjDesc;            /* Object descriptor */
MQLONG     Options;            /* Options that control the action of MQOPEN */
MQHOBJ     Hobj;               /* Object handle */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQPUT

```
MQPUT (Hconn, Hobj, &MsgDesc, &PutMsgOpts, BufferLength, Buffer, &CompCode,
&Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQHOBJ     Hobj;               /* Object handle */
MQMD       MsgDesc;            /* Message descriptor */
MQPMO      PutMsgOpts;         /* Options that control the action of MQPUT */
MQLONG     BufferLength;       /* Length in bytes of the buffer area */
MQBYTE     Buffer[n];          /* Area to contain the message data */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

MQPUT1

```
MQPUT1 (Hconn, &ObjDesc, &MsgDesc, &PutMsgOpts, BufferLength, Buffer, &CompCode,
&Reason);
```

Declare the parameters as follows:

```
MQHCONN    Hconn;              /* Connection handle */
MQOD       ObjDesc;            /* Object descriptor */
MQMD       MsgDesc;            /* Message descriptor */
MQPMO      PutMsgOpts;         /* Options that control the action of MQPUT1 */
MQLONG     BufferLength;       /* Length in bytes of the buffer area */
MQBYTE     Buffer[n];          /* Area to contain the message data */
MQLONG     CompCode;           /* Completion code */
MQLONG     Reason;             /* Reason code qualifying CompCode */
```

Elementary data types

Table 23. Elementary data types

Data Type	Representation
MQBYTE	typedef unsigned char MQBYTE;
MQBYTE24	typedef MQBYTE MQBYTE24[24];
MQBYTE32	typedef MQBYTE MQBYTE32[32];
MQCHAR	typedef char MQCHAR;
MQCHAR4	typedef MQCHAR MQCHAR4[4];
MQCHAR8	typedef MQCHAR MQCHAR8[8];
MQCHAR12	typedef MQCHAR MQCHAR12[12];
MQCHAR28	typedef MQCHAR MQCHAR28[28];
MQCHAR32	typedef MQCHAR MQCHAR32[32];
MQCHAR48	typedef MQCHAR MQCHAR48[48];
MQCHAR64	typedef MQCHAR MQCHAR64[64];
MQCHAR128	typedef MQCHAR MQCHAR128[128];
MQCHAR256	typedef MQCHAR MQCHAR256[256];
MQHCONN	typedef MQLONG MQHCONN;
MQHOBJ	typedef MQLONG MQHOBJ;
MQLONG	typedef long MQLONG;
PMQVOID	typedef void *PMQVOID;

Structure data types

MQGMO

```
typedef struct tagMQGMO {
MQCHAR4      StrucId;              /* Structure identifier */
MQLONG       Version;              /* Structure version number */
MQLONG       Options;              /* Options */
MQLONG       WaitInterval;         /* Wait interval */
MQLONG       Signal1;              /* Signal */
MQLONG       Signal2;              /* Reserved */
MQCHAR48     ResolvedQName;        /* Resolved name of destination queue */
} MQGMO;
```

MQMD

```
typedef struct tagMQMD {
MQCHAR4      StrucId;              /* Structure identifier */
MQLONG       Version;              /* Structure version number */
MQLONG       Report;               /* Report options */
MQLONG       MsgType;              /* Message type */
MQLONG       Expiry;               /* Reserved */
MQLONG       Feedback;             /* Feedback code */
MQLONG       Encoding;             /* Data encoding */
MQLONG       CodedCharSetId:       /* Coded character set identifier */
```

```
MQCHAR8      Format;              /* Format name */
MQLONG       Priority;            /* Message priority */
MQLONG       Persistence;         /* Message persistence */
MQBYTE24     MsgId;               /* Message identifier */
MQBYTE24     CorrelId;            /* Correlation identifier */
MQLONG       BackoutCount;        /* Backout counter */
MQCHAR48     ReplyToQ;            /* Name of reply-to queue */
MQCHAR48     ReplyToQMgr;         /* Name of reply queue manager */
MQCHAR12     UserIdentifier;      /* Reserved */
MQBYTE32     AccountingToken;     /* Reserved */
MQCHAR32     ApplIdentityData;    /* Reserved */
MQLONG       PutApplType;         /* Reserved */
MQCHAR28     PutApplName;         /* Reserved */
MQCHAR8      PutDate;             /* Reserved */
MQCHAR8      PutTime;             /* Reserved */
MQCHAR4      ApplOriginData;      /* Reserved */
} MQMD;
```

MQOD

```
typedef struct tagMQOD {
MQCHAR4      StrucId;             /* Structure identifier */
MQLONG       Version;             /* Structure version number */
MQLONG       ObjectType;          /* Object type */
MQCHAR48     ObjectName;          /* Object name */
MQCHAR48     ObjectQMgrName;      /* Object queue manager name */
MQCHAR48     DynamicQName;        /* Dynamic queue name */
MQCHAR12     AlternateUserId;     /* Reserved */
} MQOD;
```

MQPMO

```
typedef struct tagMQPMO {
MQCHAR4      StrucId;             /* Structure identifier */
MQLONG       Version;             /* Structure version number */
MQLONG       Options;             /* Options */
MQLONG       Timeout;             /* Reserved */
MQHOBJ       Context;             /* Reserved */
MQLONG       KnownDestCount;      /* Reserved */
MQLONG       UnknownDestCount;    /* Reserved */
MQLONG       InvalidDestCount;    /* Reserved */
MQCHAR48     ResolvedQName;       /* Resolved name of destination queue */
MQCHAR48     ResolvedQMgrName;    /* Resolved name of destination queue manager */
} MQPMO;
```

MQDLH

```
typedef struct tagMQDLH {
MQCHAR4      StrucId;             /* Structure identifier */
MQLONG       Version;             /* Structure version number */
MQLONG       Reason;              /* Reason placed on dead-letter queue */
MQCHAR48     DestQName;           /* Name of destination queue */
MQCHAR48     DestQMgrName;        /* Name of destination queue manager */
MQLONG       Encoding;            /* Data encoding */
MQLONG       CodedCharSetId;      /* Coded character set identifier */
MQCHAR8      Format;              /* Format name of application data */
MQLONG       PutApplType;         /* Application type */
MQCHAR28     PutApplName;         /* Application name */
MQCHAR8      PutDate;             /* Date put on the queue */
MQCHAR8      PutTime;             /* Time put on the queue */
} MQDLH;
```

Appendix D. Configuration worksheets

The appendix provides a set of sample worksheets presented in a format intended for duplication and use by the MQSeries System administrator or other individuals who design, configure, or require knowledge of the MQSeries System network.

The worksheets presented are:

- System List (Message Queue Manager Names)
- Application List (Queue Names & Host Systems)
- Application Look at Queues
- System Look at Queues
- Channel List
- MQSeries System Configuration (Routing Table) Work Sheet

Each of the worksheets is presented one-worksheet-per-page on the following pages. The purpose and field descriptions appear at the beginning of each worksheet. Users may use all, some, or none of these worksheets at their discretion.

Chapter 4, “Configuration” on page 29, has examples of filled-out worksheets.

System list worksheet

One list to be compiled for entire network, identifying all systems which will utilize MQSeries messaging and queuing. Each is assigned one *message_queue_manager* name.

System	Location	Hardware	QMgr_Name	Comments

Column 1 = System Name or identification (User specified terminology)

Column 2 = Location of system (City, Building, Floor, etc.)

Column 3 = Type of Hardware (Mainframe, LAN, AS/400, RS/6000, VAX, TANDEM, UNIX, etc.)

Column 4 = Assigned Message_Queue_Manager name

Column 5 = Any other user comments

Application list worksheet

One list to be compiled for entire network, identifying all applications which will utilize MQSeries messaging and queuing. Each is assigned one (or more) local *queue_name(s)* through which they will receive messages. Each is mapped to a host system from the System List worksheet.

Application	Q_Name	Hardware	QMgr_Name	Comments

Column 1 = Application Name or identification

Column 2 = Assigned local *Queue_Name*

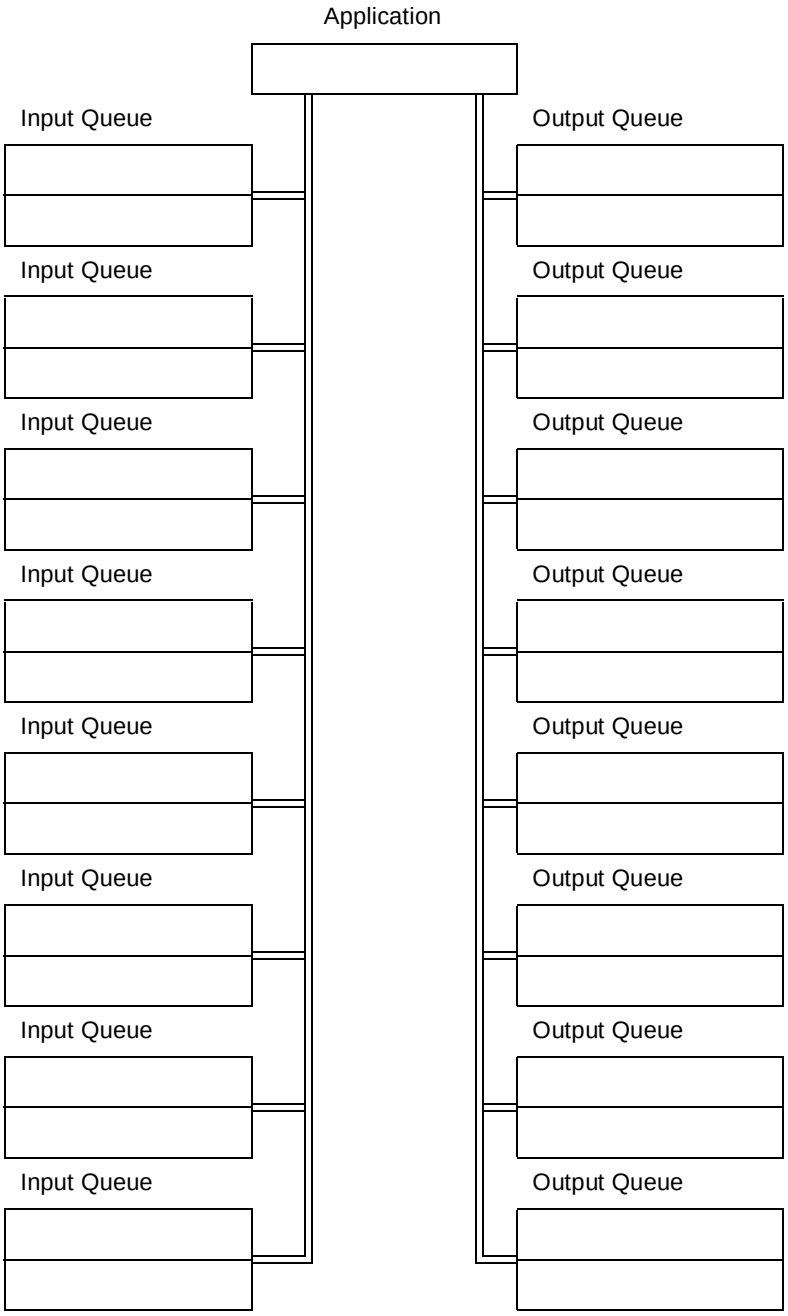
Column 3 = Type of Hardware (from SYSTEM LIST WORKSHEET)

Column 4 = Message_Queue_Manager name (from SYSTEM LIST WORKSHEET)

Column 5 = Any other user comments

Application look at queues -- worksheet

One list to be compiled for **each application**, identifying all queues with which that application will interact. (This is primary input data to applications developers.)



Each Queue Box Contains:

QUEUE_NAME

and

MESSAGE FORMAT (User Supplied Information)

System look at queues -- worksheet

One list to be compiled for **each MQSeries System**. All applications on the system are identified, all queues required on the system are identified, all channels are identified. All data is derived from previous worksheets.

Local System			Remote System	
Application	**Q_Name**	Channel	**Q_Name**	**QMgr_Name**
Input from Remote				
		<---		
		<---		
		<---		
		<---		
		<---		
Output to Remote				
		--->		
		--->		
		--->		
		--->		
		--->		
Local Messaging				
		None		
		None		
		None		
		None		
Passthru Cases *(this system is intermediate node in multi-hop routing)*				
		/---		
		\--->		
		/---		
		\--->		

Column 1 = Local Application Name or identification

Column 2 = Local *Queue_Name* for input, transmit queue name for output

Column 3 = Channel (direction of message flow)

Column 4 = Remote system Queue_Name

Column 5 = Remote system Message_Queue_Manager name

Channel list -- worksheet

One list to be compiled for entire network. All information in first 5 columns is derived from prior worksheets. All Channel names assigned (last column).

System	QMgr_Name	Channel	System	QMgr_Name	Channel_Name
		<---			
		--->			

Column 1 = System Name

Column 2 = Queue_Manager_Name for above system

Column 3 = Channel (direction of message flow)

Column 4 = System Name of “other” system in connection

Column 5 = Queue_Manager_Name for above system

Column 6 = Assigned Channel Name

MQSeries System configuration (routing table) -- worksheet

One list to be compiled for **each system**.

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
	Local			
	Local			
	Local			
	Local			
	Transmit			
	Transmit			
	Remote			
	Remote			
	Remote			

Entries are any of the following formats

Obj_Name	Type	Q_Name	QMgr_Name	Xmit_QName
Required	**Local**	n.a.	n.a.	n.a.
Required	**Transmit**	n.a.	n.a.	n.a.
Required	**Remote**	Required	Required	Optional
Required	**Alias_Q**	Required	n.a.	n.a.
Required	**Alias_M**	n.a.	Required	Optional
Required	**Alias_R**	Required	Required	n.a.

Glossary

This glossary describes terms used in this book and words used with other than their everyday meaning. In some cases, a definition may not be the only one applicable to a term, but it gives the particular sense in which the word is 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. The ANSI/EIA Standard--440-A: Fiber Optic Terminology.

Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington DC 20006. Definitions are identified by the symbol (E) after the definition. The Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

ADMINISTRATOR COMMANDS. MQSeries commands used to manage MQSeries objects, such as queues, processes and channels.

ALIAS QUEUE OBJECT. An MQSeries object, the name of which is an alias for another queue name. When an application or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the queue with the resolved name.

APAR. Authorized program analysis report.

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

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.

BROWSE. In message queuing, 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.

C

CHANNEL. See message channel.

CLIENT. The program that requests information in the particular two-program information-flow model of client/server. See also server. In an OS/2, DOS, Microsoft Windows, AIX or UNIX environment, this means a system which supports MQI application programs but does not contain the entire queue manager. For example, several client systems can all logically belong to the same queue manager.

D

DEAD-LETTER QUEUE. A queue to which a queue manager or application sends messages that it cannot deliver to their correct destination.

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

DISTRIBUTED QUEUE MANAGEMENT. In message queuing, the setup and control of message channels to queue managers on other systems.

F

FIFO. First-in-first-out.

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)

G

GET. In message queuing, to retrieve a message by removing the message from a queue or by browsing the message. See also browse.

I

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

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.

L

LOCAL DEFINITION. An MQSeries object that belongs to a local queue manager.

LOCAL DEFINITION OF A REMOTE QUEUE. An MQSeries object that belongs to a local queue manager. This object defines the attributes of a remote queue.

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. To a program, the queue manager to which the program is connected. This is the queue manager that provides message queuing services to that 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.

LOGICAL UNIT OF WORK (LUW). See unit of work.

M

MCA. Message channel agent.

MCAMD. A system program that provides a centralized channel database service allowing MCAs and MQM to access and modify the channel database.

MESSAGE. (1) In message queuing applications, a communication sent from a program to another program. (2) In system programming, information intended for the terminal operator.

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 and a communication link.

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.

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

MESSAGE QUEUE. Synonym for queue.

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

MQSERIES. A family of IBM licensed programs that provides 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. A method for communication between programs. Messaging can be synchronous or independent of time.

MQI. Message Queue Interface.

O

OBJECT. In MQSeries, objects define the attributes of queue managers, queues and process definitions.

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.

P

PERSISTENT MESSAGE. A message that survives a restart of the queue manager.

PLATFORM. In MQSeries, the operating system under which a queue manager is running. See also application environment.

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. Queues can be of type local, alias or remote. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages -- they point to other queues.

QUEUE MANAGER. (1) 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. (2) An MQSeries object that defines the attributes of a particular queue manager.

QUEUING. See message queuing.

R

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.

REMOTE QUEUE. A queue that belongs 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 is remote if it is not the queue manager to which the program is connected.

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.

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

REQUESTER CHANNEL. In MQSeries, a channel that initiates transfers, communicating with a remote server channel. The requester channel accepts messages from the server channel over a communication link and puts the messages on the local queue designated in the message.

RETURN CODES. The collective name for completion codes and reason codes.

ROLLBACK. Synonym for backout.

S

SENDER CHANNEL. In MQSeries, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver channel.

SERVER. 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 MQSeries, 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.

SYNCHRONOUS MESSAGING. A method for communication between programs in which the application waits for a reply before resuming its own processing. Contrast with time-independent 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.

T

TIME-INDEPENDENT MESSAGING. A method for communication between programs in which the requesting program proceeds with its own processing without waiting for a reply to its request. Contrast with synchronous messaging.

TRANSMISSION PROGRAM. See message channel agent.

TRANSMISSION QUEUE. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

TRIGGERING. In MQSeries, a facility that allows a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

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.

U

UNDELIVERED MESSAGE QUEUE. See dead-letter queue.

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 at a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Compare with unit of recovery.

Index

N

O

P

Q

R

S

T

V

W

Z

Sending your comments to IBM

IBM MQSeries for UnixWare

User's Guide

SC33-1379-03

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 only and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should 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, use the Readers' Comment Form
- By fax:
  - From outside the U.K., use your international access code followed by 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: WINVMD(IDRCF)
  - Internet: idrcf@winvmd.vnet.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic number to which your comment applies
- Your name/address/telephone number/fax number/network ID.

Readers' Comments

IBM MQSeries for UnixWare

User's Guide

SC33-1379-03

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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.

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or Organization	
Phone Number	

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States
Belgium	France	Israel	New Zealand	Spain	of America
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories Limited
Information Development Department (MP 095)
Hursley Park
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name _____
Company or Organization _____
Address _____
_____
EMAIL _____
Telephone _____

1 Cut along this line

4 Fasten here with adhesive tape

Program Number: 5697-265

Printed in U.S.A.

SC33-1379-03

IBM MQSeries for UnixWare **User's Guide** Version 1 Release 4.1 SC33-1379-03