MQSeries

Application Programming Reference

SC33-1673-04

MQSeries

Application Programming Reference

SC33-1673-04

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix G, “Notices” on page 571.

Fifth edition (February 1998)

This edition applies to the following products:

- MQSeries for AIX Version 5
- MQSeries for AS/400 Version 4 Release 2
- MQSeries for AT&T GIS UNIX Version 2 Release 2
- MQSeries for Digital OpenVMS Version 2 Release 2
- MQSeries for HP-UX Version 5
- MQSeries for MVS/ESA Version 1 Release 2
- MQSeries for OS/2 Warp Version 5
- MQSeries for SINIX and DC/OSx Version 2 Release 2
- MQSeries for SunOS Version 2 Release 2
- MQSeries for Sun Solaris Version 5
- MQSeries for Tandem NonStop Kernel Version 2 Release 2
- MQSeries Three Tier for OS/2 Version 1.0
- MQSeries Three Tier for AIX Version 1.0
- MQSeries for Windows NT Version 5
- MQSeries for Windows Version 2 Release 0
- MQSeries for Windows Version 2 Release 1

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

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,
England,
SO21 2JN

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

Tables

Tables

About this book

The IBM MQSeries set of products provides application programming services on various platforms that allow a new style of programming. This style enables you to code indirect program-to-program communication using *message queues*.

This book gives a full description of the MQSeries programming interface, the MQI, for the following products:

MQSeries for AIX, Version 5
MQSeries for AS/400 Version 4 Release 2
MQSeries for AT&T** GIS UNIX, Version 2 Release 2[1]
MQSeries for Digital OpenVMS Version 2 Release 2.1
MQSeries for HP-UX**, Version 5
MQSeries for MVS/ESA Version 1 Release 2
MQSeries for OS/2 Warp Version 5
MQSeries for SINIX** and DC/OSx Version 2.2
MQSeries for SunOS** Version 2.2
MQSeries for Sun Solaris** Version 5
MQSeries for Tandem NonStop Kernel Version 2 Release 2
MQSeries for Windows NT Version 5
MQSeries for Windows Version 2.0
MQSeries for Windows Version 2.1

Note: This book does not apply to the MQSeries for AS/400 Version 4 Release 2 product using the RPG programming language. You should use the *MQSeries for AS/400 Version 4 Release 2 Application Programming Reference (RPG)*, SC33-1957 for this product.

For information on how to design and write applications that use the services MQSeries provides, see the *MQSeries Application Programming Guide*.

Who this book is for

This book is for the designers of applications that use message queuing techniques, and for programmers who have to implement these designs.

What you need to know to understand this book

To write message queuing applications using MQSeries, you need to know how to write programs in one of the supported programming languages:

- C or COBOL (available on all supported platforms)
- PL/I (available on AIX, OS/2, Windows NT and MVS/ESA)
- System/390 assembler (available on MVS/ESA only)
- TAL (available on Tandem NonStop Kernel only)

If the applications you are writing are to run within a CICS system, you must also be familiar with CICS on your platform and its application programming interface.

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

To understand this book, you do not need to have written message queuing programs before.

How to use this book

This book enables you to find out quickly, for example, how to use a particular call or how to correct a particular error situation.

The book presents detailed reference information about the MQSeries programming interface, called the Message Queue Interface (MQI). It describes the:

- Data types that the MQI calls use
- Parameters and return codes for the calls
- Attributes of MQSeries objects
- Values of constants you need to use when you write MQSeries programs
- Reason codes that may occur when you run your programs

Appearance of text in this book

This book uses the following type styles:

MQOPEN	Example of the name of a call
CompCode	Example of the name of a parameter of a call, a field in a structure, or the attribute of an object
MQMD	Example of the name of a data type or structure
MQCC_FAILED	Example of the name of a constant

Terms used in this book

All new terms that this book introduces are defined in the glossary. In the body of this book, the following shortened names are used for these products:

MQSeries	The MQSeries set of products
CICS	The CICS, or Transaction Server, product for the specific platform on which you are working.

Not all of the capabilities described in this book are available in all environments. Those calls, structures, fields, or options that are not supported everywhere are identified as such in the explanatory text. Table 1 shows the short names used in this book for the various environments, and the products to which they refer.

Table 1 (Page 1 of 2). Short names used for supported environments

Short name used in this book	Full product or environment name
AIX	MQSeries for AIX Version 5.0
DOS client	MQ client applications running on PC-DOS
HP-UX	MQSeries for HP-UX Version 5.0
MVS/ESA	MQSeries for MVS/ESA Version 1.2
OpenVMS	MQSeries for Digital Open VMS Version 2.2
OS/2	MQSeries for OS/2 Warp Version 5.0
OS/400	MQSeries for AS/400 Version 4.2
Sun Solaris	MQSeries for Sun Solaris Version 5.0

Table 1 (Page 2 of 2). Short names used for supported environments	
Short name used in this book	**Full product or environment name**
Tandem NSK	MQSeries for Tandem NonStop Kernel Version 2.2
UNIX systems	The UNIX systems supported by MQSeries that are not Version 5. These are: • MQSeries for AT&T** GIS UNIX, Version 2 Release 2 • MQSeries for SINIX** and DC/OSx Version 2.2 • MQSeries for SunOS** Version 2.2
Windows client	MQ client applications running on Windows 3.1, Windows 95, or Windows NT
Windows NT	MQSeries for Windows NT Version 5.0
16-bit Windows	MQSeries for Windows Version 2.0
32-bit Windows	MQSeries for Windows Version 2.1

The following table lists the MQSeries products available for Windows, and shows the Windows platforms on which each runs.

MQSeries product	Windows 3.1	Windows 95	Windows NT
MQSeries for Windows Client	Yes	Yes	Yes
MQSeries for Windows NT	No	No	Yes
MQSeries for Windows V2.0	Yes	Yes	No
MQSeries for Windows V2.1	No	Yes	Yes

MQSeries for Windows Versions 2.0 and 2.1 support most of the features of the MQI described in this book. For information on these products, see the *MQSeries for Windows User's Guide*.

Language compilers

C++

This book does ***not*** describe the C++ programming language binding. For information on C++ you should see the *MQSeries Using C++* book.

Also, we use the following shortened names for these language compilers:

- C – see Table 2 on page xii
- COBOL – see Table 4 on page xiii
- PL/I – see Table 5 on page xiv
- Assembler/390 – see Table 6 on page xiv
- TAL – see Table 7 on page xiv

Table 2. C and C++ language compilers

Platform	Compiler
AIX	IBM C for AIX Version 3.1.4 IBM C Set++ for AIX V3.1
AIX C++	IBM C Set++ for AIX V3.1
AS/400	IBM ILE C/400 compiler (5769-CX1) for AS/400 V4R2
AS/400 C++	IBM VisualAge C++ compiler for AS/400 (5769-CX4)
AT&T	AT&T GIS High Performance C V1.0b compiler
AT&T C++	AT&T C++ language system for AT&T GIS UNIX
DC/OSx	DC/OSx C4.0 Version 4.0.1 compiler
Digital Open/VMS	DEC C Version 5.0
Digital Open/VMS C++	DEC C++ V5.0 (VAX) V5.2 (AXP)
HP-UX	C Softbench Version 5.0 HP-UX ANSI C HP C++ V3.1
HP-UX C++	HP C++ V3.1
MVS/ESA	C/370 Release 2.1.0 IBM SAA AD/Cycle C/370 Compiler
OS/2	IBM VisualAge for C++ for OS/2 V3.0 Borland C++ V2
OS/2 C++	IBM VisualAge for C++ for OS/2, V3.0
SINIX	C compiler (C-DS, MIPS) V1.1
SunOS	SPARCompiler C V3.0.1
Sun Solaris	SPARCompiler C V4.0 and V4.2
Tandem NSK	D30 or later using the WIDE memory model (32-bit integers)
Windows NT	Microsoft Visual C++ V4.0 for Windows NT IBM VisualAge for C++ for Windows V3.5
Windows NT C++	IBM VisualAge for C++ for Windows V3.5 Microsoft Visual C++ V4.0 for Windows NT
MQSeries for Windows V2.0	16-bit C - Microsoft Visual C++ V1.5
MQSeries for Windows V2.0	32-bit C - Microsoft Visual C++ V2.0
MQSeries for Windows V2.1	Microsoft Visual C++ V4.0 Borland C
DOS clients	Microsoft C/C++ V7 Microsoft Visual C++ for Windows V4.0
Windows 3.1 clients	Microsoft C/C++ V7 Microsoft Visual C++ V2.0
Windows 3.1 clients C++	Microsoft Visual C++ V1.5
Windows 95 clients	Microsoft Visual C++ V2.0
Windows 95 clients C++	IBM VisualAge for C++ V3.5 Microsoft Visual C++ V2.0
Note: AT&T has become NCR UNIX SVR4 MP-RAS, R3.0	

In addition, MQSeries for Windows V2.0 and MQSeries for Windows V2.1 support Basic compilers.

Table 3. Basic language compilers

Platform	Compiler
MQSeries for Windows V2.0 - 16-bit	Microsoft Visual Basic V3.0 or V4.0
MQSeries for Windows V2.0 - 32-bit	Microsoft Visual Basic V4.0
MQSeries for Windows V2.1	Microsoft Visual Basic V4.0

Table 4. COBOL language compilers

Platform	Compiler
AIX	The Micro Focus** COBOL compiler V3.1 and V4.0 for UNIX Systems and IBM COBOL Set for AIX Version 1.0
AS/400	IBM ILE COBOL/400 Version 4 compiler (5769-CB1) for AS/400 V4R2
Digital OpenVMS	DEC COBOL V5.0 (VAX) V2.2 (AXP)
HP-UX	COBOL Softbench Version 4.0 Micro Focus COBOL compiler Version 4.0 for UNIX Systems
MVS/ESA	VS COBOL II compiler and IBM SAA AD/Cycle COBOL/370 compiler
OS/2	Micro Focus COBOL compiler V4.0 IBM VisualAge for COBOL for OS/2 V1.1
SINIX and DC/OSx	Micro Focus COBOL compiler V3.2 for SINIX
SunOS	Micro Focus COBOL compiler V3.0
Sun Solaris	Micro Focus COBOL compiler for UNIX systems V4.0
MQSeries for Tandem NSK	D30 or later
Windows NT	Micro Focus Object COBOL compiler V3.3 or V4.0 for Windows NT Micro Focus Object COBOL compiler V3.1.J for Windows NT
DOS clients	Micro Focus COBOL V3.3
Windows 3.1 clients	Micro Focus Visual COBOL for Windows V3.3
Windows 95 clients	Micro Focus COBOL Workbench V4.0

Table 5. PL/I language compilers	
Platform	**Compiler**
AIX	IBM PL/I Set for AIX V1.1
MVS/ESA	OS PL/I Optimizing compiler IBM SAA AD/Cycle PL/I compiler
OS/2	IBM Visual Age for PL/I for OS/2 IBM PL/I for OS/2 V1.2
Windows NT	IBM Visual Age for PL/I for Windows IBM PL/I for Windows V1.2

Table 6. Assembler/390 language compilers	
Platform	**Compiler**
MVS/ESA	Assembler H assembler IBM High Level Assembler/MVS assembler

Table 7. TAL compilers	
Platform	**Compiler**
Tandem NSK	D30 or later IBM High Level Assembler/MVS assembler

MQSeries publications

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries "family" books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX V5.0
- MQSeries for AS/400 V4R2
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Digital OpenVMS V2.2
- MQSeries for HP-UX V5.0
- MQSeries for MVS/ESA V1.2
- MQSeries for OS/2 Warp V5.0
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for SunOS V2.2
- MQSeries for Sun Solaris V5.0
- MQSeries for Tandem NonStop Kernel V2.2
- MQSeries Three Tier
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.0

Any exceptions to this general rule are indicated. (Publications that support the MQSeries Level 1 products are listed in "MQSeries Level 1 product publications" on

page xviii. For a functional comparison of the Level 1 and Level 2 MQSeries products, see the *MQSeries Planning Guide*.)

MQSeries Brochure
The *MQSeries Brochure*, G511-1908, gives a brief introduction to the benefits of MQSeries. It is intended to support the purchasing decision, and describes some authentic customer use of MQSeries.

MQSeries: An Introduction to Messaging and Queuing
MQSeries: An Introduction to Messaging and Queuing, GC33-0805, describes briefly what MQSeries is, how it works, and how it can solve some classic interoperability problems. This book is intended for a more technical audience than the *MQSeries Brochure*.

MQSeries Planning Guide
The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Intercommunication
The *MQSeries Intercommunication* book, SC33-1872, defines the concepts of distributed queuing and explains how to set up a distributed queuing network in a variety of MQSeries environments. In particular, it demonstrates how to (1) configure communications to and from a representative sample of MQSeries products, (2) create required MQSeries objects, and (3) create and configure MQSeries channels. The use of channel exits is also described.

MQSeries Clients
The *MQSeries Clients* book, GC33-1632, describes how to install, configure, use, and manage MQSeries client systems.

MQSeries System Administration
The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem determination, the dead-letter queue handler, and the MQSeries links for Lotus Notes**. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.0
- MQSeries for HP-UX V5.0
- MQSeries for OS/2 Warp V5.0
- MQSeries for Sun Solaris V5.0
- MQSeries for Windows NT V5.0

MQSeries Command Reference
The *MQSeries Command Reference*, SC33-1369, contains the syntax of the MQSC commands, which are used by MQSeries system operators and administrators to manage MQSeries objects.

MQSeries Programmable System Management
The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, programmable command formats (PCFs), and installable services.

MQSeries Messages
The *MQSeries Messages* book, GC33-1876, which describes "AMQ" messages issued by MQSeries, applies to these MQSeries products only:

- MQSeries for AIX V5.0
- MQSeries for HP-UX V5.0
- MQSeries for OS/2 Warp V5.0
- MQSeries for Sun Solaris V5.0
- MQSeries for Windows NT V5.0
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

MQSeries Application Programming Guide
The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference
The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Programming Reference Summary
The *MQSeries Application Programming Reference Summary*, SX33-6095, summarizes the information in the *MQSeries Application Programming Reference* manual.

MQSeries Using C++
MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by V5.0 of MQSeries for AIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT, and by MQSeries clients supplied with those products and installed in the following environments:

- AIX
- HP-UX
- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95

MQSeries C++ is also supported by MQSeries for AS/400 V4R2.

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX

MQSeries for AIX V5.0 Quick Beginnings, GC33-1867

MQSeries for AS/400

MQSeries for AS/400 Version 4 Release 2 Licensed Program Specifications, GC33-1958

| *MQSeries for AS/400 Version 4 Release 2 Administration Guide*, GC33-1956

| *MQSeries for AS/400 Version 4 Release 2 Application Programming Reference*
| *(RPG)*, SC33-1957

MQSeries for AT&T GIS UNIX

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

MQSeries for Digital OpenVMS

MQSeries for Digital OpenVMS Version 2.2 System Management Guide, GC33-1791

MQSeries for HP-UX

MQSeries for HP-UX V5.0 Quick Beginnings, GC33-1869

MQSeries for MVS/ESA

MQSeries for MVS/ESA Version 1 Release 2 Licensed Program Specifications, GC33-1350

MQSeries for MVS/ESA Version 1 Release 2 Program Directory

MQSeries for MVS/ESA Version 1 Release 2 System Management Guide, SC33-0806

MQSeries for MVS/ESA Version 1 Release 2 Messages and Codes, GC33-0819

MQSeries for MVS/ESA Version 1 Release 2 Problem Determination Guide, GC33-0808

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp V5.0 Quick Beginnings, GC33-1868

MQSeries link for R/3

MQSeries link for R/3 Version 1.0 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

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

MQSeries for SunOS

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

MQSeries for Sun Solaris

MQSeries for Sun Solaris V5.0 Quick Beginnings, GC33-1870

| **MQSeries for Tandem NonStop Kernel**

| *MQSeries for Tandem NonStop Kernel Version 2.2 System Management*
| *Guide*, GC33-1893

MQSeries Three Tier

MQSeries Three Tier Administration Guide, SC33-1451
MQSeries Three Tier Reference Summary, SX33-6098
MQSeries Three Tier Application Design, SC33-1636
MQSeries Three Tier Application Programming, SC33-1452

MQSeries for Windows

MQSeries for Windows Version 2.0 User's Guide, GC33-1822

MQSeries for Windows Version 2.1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT V5.0 Quick Beginnings, GC33-1871

MQSeries Level 1 product publications

For information about the MQSeries Level 1 products, see the following publications:

MQSeries: Concepts and Architecture, GC33-1141

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

MQSeries for SCO UNIX Version 1.4 User's Guide, SC33-1378

MQSeries for UnixWare Version 1.4.1 User's Guide, SC33-1379

MQSeries for VSE/ESA Version 1 Release 4 Licensed Program Specifications, GC33-1483

MQSeries for VSE/ESA Version 1 Release 4 User's Guide, SC33-1142

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

BookManager format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries products, including all MQSeries V5.0 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

HTML format

The MQSeries documentation is provided in HTML format with these MQSeries products:

- MQSeries for AIX V5.0
- MQSeries for HP-UX V5.0
- MQSeries for OS/2 Warp V5.0
- MQSeries for Sun Solaris V5.0
- MQSeries for Windows NT V5.0

The MQSeries books are also available from the MQSeries product family Web site:

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

Information Presentation Facility (IPF) format

In the OS/2 environment, the MQSeries documentation is supplied in IBM IPF format on the MQSeries product CD-ROM.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows Version 2.0 and MQSeries for Windows Version 2.1.

MQSeries information available on the Internet

MQSeries web site

The MQSeries product family Web site is at:

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

| By following links from this Web site you can:

| • Obtain latest information about the MQSeries product family.
| • Access the MQSeries books in HTML format.
| • Download MQSeries SupportPacs.

Related publications

Character Data Representation Reference, SC09-1390

Summary of Changes

This section lists the major revisions to this book for the current edition and the preceding two editions.

Changes to this edition, SC33-1673-04

Changes to the book for this edition are marked by vertical bars in the left margin; these changes include:

- New versions of the following products:
  - MQSeries for AS/400
  - MQSeries for Tandem NonStop Kernel
- Minor technical and editorial improvements throughout the book

Changes to the fourth edition

Changes to the book for the fourth edition included:

- New versions of the following products:
  - MQSeries for AIX
  - MQSeries for HP-UX
  - MQSeries for OS/2
  - MQSeries for Sun Solaris
  - MQSeries for Windows NT

  The changes to the products include:
  - Addition of the MQBEGIN and MQCONNX function calls
  - Addition of the MQBO and MQCNO data type structures
  - Addition of distribution lists, which include the:
    - MQDH data type structure
    - MQOR data type structure
    - MQPMR data type structure
    - MQRR data type structure
  - Addition of message groups and segmentation of large messages
  - Addition of the MQMDE message descriptor extension data type structure
  - Addition of reference message support, which includes the MQRMH data type structure
  - Addition of PL/I language support on AIX, OS/2, and Windows NT

Changes to the third edition

Changes to the book for the third edition included:

- Addition of the MVS/ESA product
- Addition of the OS/400 V3R2 product
- Addition of the following UNIX platforms:
  - SINIX and DC/OSx
  - SunOS
  - Sun Solaris
- Addition of the following for data conversion:
  - MQCNVC function call
  - MQCONVX function call
  - MQDXP data type structure
  - Language support tables
- Addition of constants for the Windows NT platform

Chapter 1. Data type descriptions – elementary

This chapter describes the elementary data types used by the MQI.

The elementary data types are:

- MQBYTE – Byte
- MQBYTEn – String of n bytes
- MQCHAR – Single-byte character
- MQCHARn – String of n single-byte characters
- MQHCONN – Connection handle
- MQHOBJ – Object handle
- MQLONG – Long integer

Conventions used in the descriptions of data types

For each elementary data type, this chapter gives a description of its usage, in a form that is independent of the programming language. This is followed by typical declarations in each of the supported programming languages.

Elementary data types

All of the other data types described in this chapter equate either directly to these elementary data types, or to aggregates of these elementary data types (arrays or structures).

MQBYTE - Byte

The MQBYTE data type represents a single byte of data. No particular interpretation is placed on the byte—it is treated as a string of bits, and not as a binary number or character. 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 contained within it.

In the C programming language, any data type can be used for function parameters that are shown as arrays of MQBYTE. This is because such parameters are always passed by address, and in C the function parameter is declared as a pointer-to-void.

MQBYTEn – String of n bytes

Each MQBYTEn data type represents a string of *n* bytes, where *n* can take one of the following values:

16, 24, 32, or 64

Each byte is described by the MQBYTE 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 nulls to fill the string.

When the queue manager returns byte strings to the application (for example, on the MQGET call), the queue manager always pads with nulls to the defined length of the string.

Constants are available that define the lengths of byte string fields; see "MQ_★ (Lengths of character string and byte fields)" on page 449.

MQCHAR – character

The MQCHAR data type represents a single character. The coded character set identifier of the character is that of the queue manager (see the *CodedCharSetId* attribute on page 372). No special alignment is required.

Note: Application message data specified on the MQGET, MQPUT, and MQPUT1 calls is described by the MQBYTE data type, not the MQCHAR 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, or 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; the null character and characters following it are treated as blanks, up to the defined length of the string. The places where a null can 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; the queue manager does not use the null character to delimit the string.

Constants are available that define the lengths of character string fields; see "MQ_★ (Lengths of character string and byte fields)" on page 449.

MQHCONN – Connection handle

The MQHCONN data type represents a connection handle, that is, the connection to a particular queue manager. A connection handle must be aligned on its natural boundary.

Note: Applications must test variables of this type for equality only.

MQHOBJ – Object handle

The MQHOBJ data type represents an object handle that gives access to an object. An object handle must be aligned on its natural boundary.

Note: Applications must test variables of this type for equality only.

MQLONG – Long integer

The MQLONG data type is a 32-bit signed binary integer that can take any value in the range –2 147 483 648 through +2 147 483 647, unless otherwise restricted by the context. For COBOL, the valid range is limited to –999 999 999 through +999 999 999. An MQLONG must be aligned on its natural boundary.

Elementary data types – C programming language

Table 8. Elementary data types in C

Data type	Representation
MQBYTE	`typedef unsigned char MQBYTE;`
MQBYTE16	`typedef MQBYTE MQBYTE16[16];`
MQBYTE24	`typedef MQBYTE MQBYTE24[24];`
MQBYTE32	`typedef MQBYTE MQBYTE32[32];`
MQBYTE64	`typedef MQBYTE MQBYTE64[64];`
MQCHAR	`typedef char MQCHAR;`
MQCHAR4	`typedef MQCHAR MQCHAR4[4];`
MQCHAR8	`typedef MQCHAR MQCHAR8[8];`
MQCHAR12	`typedef MQCHAR MQCHAR12[12];`
MQCHAR16	`typedef MQCHAR MQCHAR16[16];`
MQCHAR20	`typedef MQCHAR MQCHAR20[20];`
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;`
MQPTR	`typedef void MQPOINTER MQPTR;`
PMQLONG	`typedef MQLONG MQPOINTER PMQLONG;`

| See “Data types” on page 9 for a description of the MQPOINTER macro variable.

Elementary data types - COBOL programming language

Table 9 (Page 1 of 2). Elementary data types in COBOL

Data type	Representation
MQBYTE	`PIC X`
MQBYTE16	`PIC X(16)`
MQBYTE24	`PIC X(24)`
MQBYTE32	`PIC X(32)`
MQBYTE64	`PIC X(64)`
MQCHAR	`PIC X`
MQCHAR4	`PIC X(4)`

Table 9 (Page 2 of 2). Elementary data types in COBOL

Data type	Representation
MQCHAR8	PIC X(8)
MQCHAR12	PIC X(12)
MQCHAR16	PIC X(16)
MQCHAR20	PIC X(20)
MQCHAR28	PIC X(28)
MQCHAR32	PIC X(32)
MQCHAR48	PIC X(48)
MQCHAR64	PIC X(64)
MQCHAR128	PIC X(128)
MQCHAR256	PIC X(256)
MQHCONN	PIC S9(9) BINARY
MQHOBJ	PIC S9(9) BINARY
MQLONG	PIC S9(9) BINARY
MQPTR	POINTER
PMQLONG	POINTER

Elementary data types – PL/I language

Table 10. Elementary data types in PL/I

Data type	Representation
MQBYTE	char(1)
MQBYTE16	char(16)
MQBYTE24	char(24)
MQBYTE32	char(32)
MQBYTE64	char(64)
MQCHAR	char(1)
MQCHAR4	char(4)
MQCHAR8	char(8)
MQCHAR12	char(12)
MQCHAR16	char(16)
MQCHAR20	char(20)
MQCHAR28	char(28)
MQCHAR32	char(32)
MQCHAR48	char(48)
MQCHAR64	char(64)
MQCHAR128	char(128)
MQCHAR256	char(256)
MQHCONN	fixed bin(31)
MQHOBJ	fixed bin(31)
MQLONG	fixed bin(31)
PMQLONG	pointer

Elementary data types – System/390 Assembler (MVS/ESA only)

Table 11. Elementary data types in System/390 assembler

Data type	Representation
MQBYTE	DS XL1
MQBYTE16	DS XL16
MQBYTE24	DS XL24
MQBYTE32	DS XL32
MQBYTE64	DS XL64
MQCHAR	DS CL1
MQCHAR4	DS CL4
MQCHAR8	DS CL8
MQCHAR12	DS CL12
MQCHAR16	DS CL16
MQCHAR20	DS CL20
MQCHAR28	DS CL28
MQCHAR32	DS CL32
MQCHAR48	DS CL48
MQCHAR64	DS CL64
MQCHAR128	DS CL128
MQCHAR256	DS CL256
MQHCONN	DS F
MQHOBJ	DS F
MQLONG	DS F
PMQLONG	DS F

Elementary data types – TAL programming language

Table 12. Elementary data types in TAL

Data Type	Representation
MQBYTE	STRING
MQBYTE24	BEGIN STRING BYTE [0:23];END
MQBYTE32	BEGIN STRING BYTE [0:31];END
MQCHAR	STRING
MQCHAR4	BEGIN STRING BYTE [0:3];END
MQCHAR8	BEGIN STRING BYTE [0:7]; END
MQCHAR12	BEGIN STRING BYTE [0:11];END
MQCHAR28	BEGIN STRING BYTE [0:27];END
MQCHAR32	BEGIN STRING BYTE [0:31];END
MQCHAR48	BEGIN STRING BYTE [0:47];END
MQCHAR64	BEGIN STRING BYTE [0:63];END
MQCHAR128	BEGIN STRING BYTE [0:127];END
MQCHAR256	BEGIN STRING BYTE [0:255];END
MQHCONN	INT(32)
MQHOBJ	INT(32)
MQLONG	INT(32)

Chapter 2. Data type descriptions – structures

This chapter describes the structure data types used by the MQI, which are:

- MQBO – Begin options
- MQCNO – Connect options
- MQGMO – Get-message options
- MQMD – Message descriptor
- MQMDE – Message descriptor extension
- MQOD – Object descriptor
- MQOR – Object record
- MQPMO – Put-message options
- MQPMR – Put message record
- MQRR – Response record

The MQI also uses the following structure data types, which are included in this chapter for completeness, but they are not part of the application programming interface.

- MQCIH – CICS bridge header (MVS/ESA only)
- MQDH – Distribution header
- MQDLH – Dead-letter (undelivered-message) header
- MQIIH – IMS bridge header (MVS/ESA only)
- MQTM – Trigger message
- MQTMC2 – Trigger message (character format 2)
- MQXQH – Transmission queue header

Note: The MQDXP – data conversion exit parameter structure is in Appendix D, "Data-conversion" on page 495, together with the associated data conversion calls.

Conventions used in the descriptions of data types

For each structure data type, this chapter gives a description of its usage, in a form that is independent of the programming language. This is followed by typical declarations in each of the supported programming languages.

The description of each structure data type contains the following sections:

Structure name
The name of the structure, followed by a brief description of the purpose of the structure.

Fields
For each field, the name is followed by its elementary data type in parentheses ( ); for example:

Version (MQLONG)

There is also a description of the purpose of the field, together with a list of any values that the field can take. Names of constants are shown in uppercase; for example, MQGMO_STRUC_ID. A set of constants having the same prefix is shown using the ⋆ character, for example: MQIA_⋆.

In the descriptions of the fields, the following terms are used:

input You supply information in the field when you make a call.

output The queue manager returns information in the field when the call completes or fails.

input/output You supply information in the field when you make a call, and the queue manager changes the information when the call completes or fails.

Initial values
A table showing the initial values for each field in the data definition files supplied with the MQI.

C declaration
Typical declaration of the structure in C.

COBOL declaration
Typical declaration of the structure in COBOL.

PL/I declaration
Typical declaration of the structure in PL/I[2].

System/390 assembler-language declaration
Typical declaration of the structure in System/390[2] assembler language.

Language considerations

This section outlines the requirements for data types in the following programming languages:

- C – see "Using the data types in the C programming language"
- COBOL – see "Using the data types in the COBOL programming language" on page 12
- PL/I – see "Using the data types in the PL/I programming language" on page 15
- Assembler/390 – see "Using the data types in the System/390 Assembler programming language" on page 16

Using the data types in the C programming language

This section contains information to help you use the MQI from the C programming language.

Header files

Header files are provided as part of the definition of the message queue interface, to assist with the writing of C application programs that use message queuing. These header files are summarized in Table 13 on page 9.

[2] PL/I and assembler are not sensitive to case, so the names of calls, structure fields, and constants can be coded in lowercase, uppercase, or mixed case.

Table 13. C header file	
Filename	**Contents**
CMQC	Function prototypes, data types, and named constants for the main MQI
CMQXC	Function prototypes, data types, and named constants for the data-conversion exit

To improve the portability of applications, it is recommended that the name of the header file should be coded in lowercase on the **#include** preprocessor directive:

```
#include "cmqc.h"
```

Functions

Parameters that are *input-only* and of 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.

No parameter is returned as the value of the function; in C terminology, this means that all functions return **void**.

The attributes of the function are defined by the MQENTRY macro variable; the value of this macro variable depends on the environment.

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 perfectly valid to declare the parameters in this way, but it is usually more convenient to declare them as the particular structure which describes the layout of the data in the message. The actual function parameter is declared as a pointer-to-void, and so 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 name of the elementary or structure data type prefixed with the letter "P" to denote a pointer. The attributes of the pointer are defined by the MQPOINTER macro variable; the value of this macro variable depends on the environment. The following illustrates how pointer datatypes are declared:

```
#define MQPOINTER *               /* depends on environment */
...
typedef MQLONG MQPOINTER PMQLONG;  /* pointer to MQLONG */
typedef MQMD   MQPOINTER 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 a 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.

Do not use the string functions, which require the string to be terminated by a null (**strcpy**, **strcmp**, **strcat**). Also, do not use the function **strlen** to determine the length of the string; use instead the **sizeof** function to determine the length of the field.

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};
```

For some character fields (for example, the *StrucId* fields which occur in most structures, or the *Format* field which occurs in MQMD), the MQI defines particular values that are valid. For each of the valid values, *two* macro variables are provided:

- One macro variable defines the value as a string whose length excluding the implied null matches exactly the defined length of the field. For example, for the *Format* field in MQMD the following macro variable is provided (the symbol "b" represents a blank character):

  ```
  #define MQFMT_STRING "MQSTRbbb"
  ```

  Use this form with the **memcpy** and **memcmp** functions.

- The other macro variable defines the value as an array of characters; the name of this macro variable is the name of the string form suffixed with "_ARRAY". For example:

  ```
  #define MQFMT_STRING_ARRAY 'M','Q','S','T','R',' ',' ',' '
  ```

  Use this form to initialize the field when an instance of the structure is declared with values different from those provided by the MQMD_DEFAULT macro variable.[3]

Initial values for dynamic structures

When a variable number of instances of a structure is required, the instances are usually created in main storage obtained dynamically using the **calloc** or **malloc** functions. To initialize the fields in such structures, the following technique is recommended:

1. Declare an instance of the structure using the appropriate MQXXX_DEFAULT macro variable to initialize the structure. This instance becomes the "model" for other instances:

   ```
   MQMD Model = {MQMD_DEFAULT};  /* declare model instance */
   ```

   The **static** or **auto** keywords can be coded on the declaration in order to give the model instance static or dynamic lifetime, as required.

2. Use the **calloc** or **malloc** functions to obtain storage for a dynamic instance of the structure:

   ```
   PMQMD Instance;
   Instance = malloc(sizeof(MQMD));  /* get storage for dynamic instance */
   ```

3. Use the **memcpy** function to copy the model instance to the dynamic instance:

   ```
   memcpy(Instance,&Model,sizeof(MQMD));  /* initialize dynamic instance */
   ```

Use from C++

For the C++ programming language, the header files contain the following additional statements that are included only when a C++ compiler is used:

```
#ifdef __cplusplus
  extern "C" {
#endif

/* rest of header file */

#ifdef __cplusplus
  }
#endif
```

Notational conventions

The sections that follow show how the:

- Calls should be invoked
- Parameters should be declared
- Various data types should be declared.

---

[3] This is not always necessary; in some environments the string form of the value can be used in both situations. However, the array form is recommended for declarations, since this is required for compatibility with the C++ programming language.

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.

Using the data types in the COBOL programming language

This section contains information to help you use the MQI from the COBOL programming language.

COPY files

Various COPY files are provided as part of the definition of the message queue interface, to assist with the writing of COBOL application programs that use message queuing. There are two files containing the named constants, and two files for each of the structures.

Each structure is provided in two forms: a form with initial values, and a form without.

- The structures with initial values can be used in the **WORKING-STORAGE SECTION** of a COBOL program, and are contained in COPY files which have names suffixed with the letter “V” (mnemonic for “Values”).
- The structures without initial values can be used in the **LINKAGE SECTION** of a COBOL program, and are contained in COPY files which have names suffixed with the letter “L” (mnemonic for “Linkage”).

The COPY files are summarized in Table 14.

Table 14 (Page 1 of 2). COBOL COPY files

File name (with initial values)	File name (without initial values)	Contents
CMQBOV	CMQBOL	Begin options structure
CMQCNOV	CMQCNOL	Connect options structure
CMQDHV	CMQDHL	Distribution header structure
CMQDLHV	CMQDLHL	Dead-letter (undelivered-message) header structure
CMQDXPV	CMQDXPL	Data-conversion-exit parameter structure
CMQGMOV	CMQGMOL	Get-message options structure
CMQIIHV	CMQIIHL	IMS information header structure
CMQMDV	CMQMDL	Message descriptor structure
CMQMDEV	CMQMDEL	Message descriptor extension structure
CMQODV	CMQODL	Object descriptor structure
CMQORV	CMQORL	Object record structure
CMQPMOV	CMQPMOL	Put-message options structure
CMQPMRV	CMQPMRL	Put-message record structure
CMQRRV	CMQRRL	Response record structure
CMQTMV	CMQTML	Trigger-message structure
–	CMQTMCL	Trigger-message structure (character format)

Table 14 (Page 2 of 2). COBOL COPY files		
File name (with initial values)	**File name (without initial values)**	**Contents**
CMQXQHV	CMQXQHL	Transmission-queue header structure
CMQV	–	Named constants for main MQI
CMQXV	–	Named constants for data-conversion exit

Structures

In the COPY file, each structure declaration begins with a level-10 item; this enables several instances of the structure to be declared, by coding the level-01 declaration and then using the **COPY** statement to copy in the remainder of the structure declaration. To reference the appropriate instance, the **IN** keyword can be used:

```
* Declare two instances of MQMD
 01  MY-MQMD.
     COPY CMQMDV.
 01  MY-OTHER-MQMD.
     COPY CMQMDV.
*
* Set MSGTYPE field in MY-OTHER-MQMD
     MOVE MQMT-REQUEST TO MQMD-MSGTYPE IN MY-OTHER-MQMD.
```

The structures should be aligned on 4-byte boundaries. If the **COPY** statement is used to include a structure following an item which is not the level-01 item, try to ensure that the structure is a multiple of 4-bytes from the start of the level-01 item; failure to do this may result in a performance degradation.

In Chapter 1, “Data type descriptions – elementary” on page 1, the names of fields in structures are shown without a prefix. In COBOL, the field names are prefixed with the name of the structure followed by a hyphen. However, if the structure name ends with a numeric digit, indicating that the structure is a second or later version of the original structure, the numeric digit is *omitted* from the prefix. Field names in COBOL are shown in uppercase (although mixed case or lowercase can be used if required). For example, the field *MsgType* described on page 112 becomes MQMD-MSGTYPE in COBOL.

The V-suffix structures are declared with initial values for all of the fields, and so it is necessary to set only those fields where the value required is different from the initial value.

Pointers

Some structures need to address optional data that may be discontiguous with the structure. For example, the MQOR and MQRR records addressed by the MQOD structure are like this. To address this optional data, the structures contain fields that are declared with the pointer data type. However, COBOL does not support the pointer data type in all environments. Because of this, the optional data can also be addressed using fields which contain the offset of the data from the start of the structure.

If an application is intended to be portable between environments, the application designer should ascertain whether the pointer data type is available in all of the

intended environments. If it is not, the application should address the optional data using the offset fields instead of the pointer fields.

In those environments where pointers are not supported, the pointer fields are declared as byte strings of the appropriate length, with the initial value being the all-null byte string. This initial value should not be altered if the offset fields are being used.

Named constants

In this book, the names of constants are shown containing the underscore character (_) as part of the name. In COBOL, the hyphen character (-) must be used in place of the underscore.

Constants which have character-string values use the single-quote character as the string delimiter ('). In some environments it may be necessary to specify an appropriate compiler option to cause the compiler to accept the single quote as the string delimiter.

The named constants are declared in the COPY files as level-10 items. To use the constants, the level-01 item must be declared explicitly, and then the **COPY** statement used to copy in the declarations of the constants:

```
* Declare a structure to hold the constants
 01  MY-MQ-CONSTANTS.
     COPY CMQV.
```

The above method causes the constants to occupy storage in the program even if they are not referenced. If the constants are included in many separate programs within the same run unit, multiple copies of the constants will exist; this may result in a significant amount of main storage being consumed. This can be avoided by using one of the following techniques:

- Add the **GLOBAL** clause to the level-01 declaration:

  ```
  * Declare a global structure to hold the constants
   01  MY-MQ-CONSTANTS GLOBAL.
       COPY CMQV.
  ```

  This causes storage to be allocated for only *one* set of constants within the run unit; the constants, however, can be referenced by *any* program within the run unit, not just the program which contains the level-01 declaration.

  Note: The **GLOBAL** clause is not supported in all environments.

- Manually copy into each program only those constants that are referenced by that program; do not use the **COPY** statement to copy all of the constants into the program.

Notational conventions

The sections that follow show how the:

- Calls should be invoked
- Parameters should be declared
- Various data types should be declared

In a number of cases, parameters are tables or character strings 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.

Using the data types in the PL/I programming language

This section contains information to help you use the MQI from the PL/I programming language.

INCLUDE files

Two INCLUDE files are provided as part of the definition of the message queue interface, to assist with the writing of PL/I application programs that use message queuing. There is one INCLUDE file containing the structures and named constants, and one containing the entry-point declarations. These files are summarized in Table 15.

Table 15. PL/I INCLUDE file

Filename	Contents
CMQEPP	Entry points
CMQP	Structures, named constants

To improve the portability of applications, it is recommended that the names of the INCLUDE files should be coded in lowercase on the **%include** compiler directive:

```
%include syslib(cmqp);
%include syslib(cmqepp);
```

Structures

Structures are declared with the **BASED** attribute, and so do not occupy any storage unless the program declares one or more instances of a structure.

An instance of a structure can be declared by using the **LIKE** attribute:

```
%include syslib(cmqp);
%include syslib(cmqepp);

dcl 1 my_mqmd       like MQMD;  /* one instance */
dcl 1 my_other_mqmd like MQMD;  /* another one */
```

The structure fields are declared with the **INITIAL** attribute. When the **LIKE** attribute is used to declare an instance of a structure, that instance inherits the initial values defined for that structure. Thus it is necessary to set only those fields where the value required is different from the initial value.

PL/I is not sensitive to case, and so the names of calls, structure fields, and constants can be coded in lowercase, uppercase, or mixed case.

Named constants

The named constants are declared as macro variables; as a result, named constants which are not referenced by the program do not occupy any storage in the compiled procedure. However, the compiler option which causes the source to be processed by the macro preprocessor must be specified when the program is compiled.

All of the macro variables are character variables, even the ones which represent numeric values. Although this may seem counter-intuitive, it does not result in any data-type conflict after the macro variables have been substituted by the macro processor:

```
%dcl MQMD_STRUC_ID char;
%MQMD_STRUC_ID = '''MD  ''';

%dcl MQMD_VERSION_1 char;
%MQMD_VERSION_1 = '1';
```

Notational conventions

The sections following show how the:

- Calls should be invoked
- Parameters should be declared
- Various data types should be declared.

In a number of cases, parameters are arrays or character strings 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.

Using the data types in the System/390 Assembler programming language

This section contains information to help you use the MQI from the System/390 Assembler programming language.

Macros

Various macros are provided as part of the definition of the message queue interface, to assist with the writing of assembler application programs that use message queuing. There is one macro for the named constants, and one macro for each of the structures. These files are summarized in Table 16.

Table 16. Assembler macros

Filename	Contents
CMQA	Named constants (“equates”)
CMQDLHA	Dead-letter header structure
CMQDXPA	Data-conversion exit parameter structure
CMQGMOA	Get-message options structure
CMQIIHA	IMS bridge structure
CMQMDA	Message descriptor structure
CMQODA	Object descriptor structure
CMQPMOA	Put-message options structure
CMQTMA	Trigger message structure
CMQXQHA	Transmission-queue header structure

Names

In this book, the names of parameters and the names of fields in structures, are shown in a mixture of upper and lowercase. In assembler, all names must be coded in uppercase.

Structures

The structures are generated by macros that have various parameters to control the action of the macro.

Specifying the name of the structure: To allow more than one instance of a structure to be declared, the macro prefixes the name of each field in the structure with a user-specifiable string and an underscore. The string used is the label specified on the invocation of the macro. If no label is specified, the name of the structure is used to construct the prefix:

```
* Declare two object descriptors
     CMQODA                          Prefix used="MQOD_" (the default)
MY_MQOD  CMQODA                          Prefix used="MY_MQOD_"
```

The structure declarations use the default prefix.

Specifying the form of the structure: Structure declarations can be generated by the macro in one of two forms, controlled by the **DSECT** parameter:

DSECT=YES An assembler **DSECT** instruction is used to start a new data section; the structure definition immediately follows the **DSECT** statement. The label on the macro invocation is used as the name of the data section; if no label is specified, the name of the structure is used.

DSECT=NO Assembler **DC** instructions are used to define the structure at the current position in the routine. The fields are initialized with values, which can be specified by coding the relevant parameters on the macro invocation. Fields for which no values are specified on the macro invocation are initialized with default values.

DSECT=NO is assumed if the **DSECT** parameter is not specified.

Declaring one structure embedded within another: To declare one structure as a component of another structure, the **NESTED** parameter should be used:

NESTED=YES The structure declaration is nested within another.

NESTED=NO The structure declaration is not nested within another.

NESTED=NO is assumed if the **NESTED** parameter is not specified.

Controlling the listing: The appearance of the structure declaration in the assembler listing can be controlled by means of the **LIST** parameter:

LIST=YES The structure declaration appears in the assembler listing.

LIST=NO The structure declaration does not appear in the assembler listing.

LIST=NO is assumed if the **LIST** parameter is not specified.

Specifying initial values for fields: The value to be used to initialize a field in a structure can be specified by coding the name of that field (without the prefix) as a parameter on the macro invocation, accompanied by the value required. For example, to declare a message-descriptor structure with the *MsgType* field initialized with MQMT_REQUEST, and the *ReplyToQ* field initialized with the string "MY_REPLY_TO_QUEUE", the following could be used:

```
MY_MQMD        CMQMDA          MSGTYPE=MQMT_REQUEST,                X
                             REPLYTOQ=MY_REPLY_TO_QUEUE
```

If a named constant (equate) is specified as a value on the macro invocation, the CMQA macro must be used in order to define the named constant. Values which are character strings must not be enclosed in single quotes.

Notational conventions

The sections that follow show how the:

- Calls should be invoked
- Parameters should be declared
- Various data types should be declared.

In the sample declarations of the elementary datatypes, the string "var" is used to represent the name of a variable; when that declaration is coded, "var" must be replaced by the actual name required.

In a number of cases, parameters are arrays or character strings 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.

Structure data types

Programming languages vary in their level of support for structures, and certain rules and conventions are adopted in order to allow the MQI structures to be mapped consistently in each programming language:

1. Structures are aligned on their natural boundaries. All MQI structures require 4-byte alignment.
2. Each field in the structure is aligned on its natural boundary. Fields with data types that equate to 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 alignment. All MQI structures have lengths that are multiples of 4 bytes.
4. Where necessary, padding fields are declared explicitly to ensure compliance with rules 2 and 3 above.

MQBO – Begin options

The following table summarizes the fields in the structure.

Table 17. Fields in MQBO

Field	Description	Page
StrucId	Structure identifier	19
Version	Structure version number	19
Options	Options that control the action of MQBEGIN	19

The MQBO structure is an input/output parameter for the MQBEGIN call.

This structure is supported in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQBO_STRUC_ID
Identifier for begin-options structure.

For the C programming language, the constant MQBO_STRUC_ID_ARRAY is also defined; this has the same value as MQBO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQBO_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQBO_VERSION_1
Version number for begin-options structure.

The following constant specifies the version number of the current version:

MQBO_CURRENT_VERSION
Current version of begin-options structure.

This is always an input field. The initial value of this field is MQBO_VERSION_1.

Options (MQLONG)
Options that control the action of MQBEGIN.

The value must be:

MQBO_NONE
No options specified.

This is always an input field. The initial value of this field is MQBO_NONE.

Table 18. Initial values of fields in MQBO

Field name	Name of constant	Value of constant
StrucId	MQBO_STRUC_ID	'BObb' (See note 1)
Version	MQBO_VERSION_1	1
Options	MQBO_NONE	0

Notes:

1. The symbol 'b' represents a single blank character.
2. In the C programming language, the macro variable MQBO_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQBO MyBO = {MQBO_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQBO {
  MQCHAR4  StrucId;  /* Structure identifier */
  MQLONG   Version;  /* Structure version number */
  MQLONG   Options;  /* Options that control the action of MQBEGIN */
 } MQBO;
```

COBOL language declaration

```
**   MQBO structure
  10 MQBO.
**    Structure identifier
   15 MQBO-STRUCID PIC X(4).
**    Structure version number
   15 MQBO-VERSION PIC S9(9) BINARY.
**    Options that control the action of MQBEGIN
   15 MQBO-OPTIONS PIC S9(9) BINARY.
```

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

```
dcl
 1 MQBO based,
  3 StrucId char(4),       /* Structure identifier */
  3 Version fixed bin(31), /* Structure version number */
  3 Options fixed bin(31); /* Options that control the action of
                              MQBEGIN */
```

MQCIH – CICS bridge header (MVS/ESA only)

The following table summarizes the fields in the structure.

Table 19. Fields in MQCIH

Field	Description	Page
StrucId	Structure identifier	22
Version	Structure version number	22
StrucLength	Length of MQCIH structure	22
Format	MQ format name	23
ReturnCode	Return code from bridge	23
CompCode	MQ completion code or CICS EIBRESP	24
Reason	MQ reason or feedback code, or CICS EIBRESP2	24
UOWControl	Unit-of-work control	24
GetWaitInterval	Wait interval for MQGET call issued by bridge task	25
LinkType	Link type	25
OutputDataLength	Output COMMAREA data length	25
Function	MQ call name or CICS EIBFN function	26
AbendCode	Abend code	27
Authenticator	Password or passticket	27
ReplyToFormat	MQ format name of reply message	27

The MQCIH structure describes the information that can be present at the start of a message sent to the CICS bridge through MQSeries for MVS/ESA. The structure can be omitted if the values required by the application are the same as the initial values shown in Table 21 on page 29. The format name of this structure is MQFMT_CICS.

This structure is supported on MVS/ESA only.

Special conditions apply to the character set and encoding used for the MQCIH structure and application message data:

- Applications that connect to the queue manager which owns the CICS bridge queue must provide an MQCIH structure that is in the character set and encoding of the queue manager. This is because data conversion of the MQCIH structure is not performed in this case.
- Applications that connect to other queue managers can provide an MQCIH structure that is in any of the supported character sets and encodings; conversion of the MQCIH and application message data is performed by the queue manager as necessary.

  Note: There is one exception to this. If the queue manager which owns the CICS bridge queue is using CICS for distributed queuing, the MQCIH must be in the character set and encoding of that queue manager.
- The application message data following the MQCIH structure must be in the same character set and encoding as the MQCIH structure. The

CodedCharSetId and *Encoding* fields in the MQCIH structure cannot be used to specify the character set and encoding of the application message data.

Error information is returned in the *ReturnCode*, *Function*, *CompCode*, *Reason*, and *AbendCode* fields. Which of them is set depends on the value of the *ReturnCode* field; see Table 20.

Table 20. Contents of error information fields in MQCIH structure

ReturnCode	*Function*	*CompCode*	*Reason*	*AbendCode*
MQCRC_OK	–	–	–	–
MQCRC_BRIDGE_ERROR	–	–	MQFB_CICS_★	–
MQCRC_MQ_API_ERROR MQCRC_BRIDGE_TIMEOUT	MQ call name	MQ *CompCode*	MQ *Reason*	–
MQCRC_CICS_EXEC_ERROR MQCRC_SECURITY_ERROR MQCRC_PROGRAM_NOT_AVAILABLE MQCRC_TRANSID_NOT_AVAILABLE	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	–
MQCRC_BRIDGE_ABEND MQCRC_APPLICATION_ABEND	–	–	–	CICS ABCODE

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQCIH_STRUC_ID
Identifier for CICS information header structure.

For the C programming language, the constant MQCIH_STRUC_ID_ARRAY is also defined; this has the same value as MQCIH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQCIH_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQCIH_VERSION_1
Version number for CICS information header structure.

The following constant specifies the version number of the current version:

MQCIH_CURRENT_VERSION
Current version of CICS information header structure.

The initial value of this field is MQCIH_VERSION_1.

StrucLength (MQLONG)
Length of MQCIH structure.

The value must be:

MQCIH_LENGTH_1
Length of CICS information header structure.

The initial value of this field is MQCIH_LENGTH_1.

Encoding (MQLONG)
Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

CodedCharSetId (MQLONG)
Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

Format (MQCHAR8)
MQ format name.

This is the MQ format name of the application message data which follows the MQCIH structure. The rules for coding this are the same as those for the *Format* field in MQMD.

This format name is also used for the reply message, if the *ReplyToFormat* field has the value MQFMT_NONE.

If the request message results in the generation of an error reply message, the error reply message has a format name of MQFMT_STRING.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

Flags (MQLONG)
Reserved.

The value must be:

MQCIH_NONE
No flags.

The initial value of this field is MQCIH_NONE.

ReturnCode (MQLONG)
Return code from bridge.

This is the return code from the CICS bridge describing the outcome of the processing performed by the bridge. The *Function*, *CompCode*, *Reason*, and *AbendCode* fields may contain additional information (see Table 20 on page 22). The value is one of the following:

MQCRC_APPLICATION_ABEND
(5, X'005') Application ended abnormally.

MQCRC_BRIDGE_ABEND
(4, X'004') CICS bridge ended abnormally.

MQCRC_BRIDGE_ERROR
(3, X'003') CICS bridge detected an error.

MQCRC_BRIDGE_TIMEOUT
(8, X'008') Second or later message within current unit of work not received within specified time.

MQCRC_CICS_EXEC_ERROR
(1, X'001') EXEC CICS statement detected an error.

MQCRC_MQ_API_ERROR
(2, X'002') MQ call detected an error.

MQCRC_OK
(0, X'000') No error.

MQCRC_PROGRAM_NOT_AVAILABLE
(7, X'007') Program not available.

MQCRC_SECURITY_ERROR
(6, X'006') Security error occurred.

MQCRC_TRANSID_NOT_AVAILABLE
(9, X'009') Transaction not available.

The initial value of this field is MQCRC_OK.

CompCode (MQLONG)
MQ completion code or CICS EIBRESP.

The value returned in this field is dependent on *ReturnCode*; see Table 20 on page 22.

The initial value of this field is MQCC_OK

Reason (MQLONG)
MQ reason or feedback code, or CICS EIBRESP2.

The value returned in this field is dependent on *ReturnCode*; see Table 20 on page 22.

The initial value of this field is MQRC_NONE.

UOWControl (MQLONG)
Unit-of-work control.

This controls the unit-of-work processing performed by the CICS bridge. The field indicates whether the CICS bridge should start a unit of work, perform the requested function within the current unit of work, or end the unit of work by committing it or backing it out. Various combinations are supported, to optimize the data transmission flows.

The value must be one of the following:

MQCUOWC_FIRST
Start unit of work and perform function.

MQCUOWC_MIDDLE
Perform function within current unit of work.

MQCUOWC_LAST
Perform function, then commit the unit of work.

MQCUOWC_ONLY
Start unit of work, perform function, then commit the unit of work.

MQCUOWC_COMMIT
Commit the unit of work.

MQCUOWC_BACKOUT
Back out the unit of work.

The initial value of this field is MQCUOWC_ONLY.

GetWaitInterval (MQLONG)
Wait interval for MQGET call issued by bridge task.

This field is applicable only when *UOWControl* has the value MQCUOWC_FIRST. It allows the sending application to specify the approximate time in milliseconds that the MQGET calls issued by the bridge should wait for second and subsequent request messages for the unit of work started by this message. This overrides the default wait interval used by the bridge. The following special values may be used:

MQCGWI_DEFAULT
Default wait interval.

This causes the CICS bridge to wait for the period of time specified when the bridge was started.

MQWI_UNLIMITED
Unlimited wait interval.

The initial value of this field is MQCGWI_DEFAULT.

LinkType (MQLONG)
Link type.

This indicates the type of object that the bridge should try to link. The value must be:

MQCLT_PROGRAM
Program.

The initial value of this field is MQCLT_PROGRAM.

OutputDataLength (MQLONG)
Output COMMAREA data length.

This is the length of the user data to be returned to the client in a reply message. This length includes the 8-byte program name. The length of the COMMAREA passed to the linked program is the maximum of this field and the length of the user data in the request message, minus 8.

Note: The length of the user data in a message is the length of the message *excluding* the MQCIH structure.

If the length of the user data in the request message is smaller than *OutputDataLength*, the DATALENGTH option of the LINK command is used; this allows the LINK to be function-shipped efficiently to another CICS region.

The following special value may be used:

MQCODL_AS_INPUT
Output length is same as input length.

This value may be needed even if no reply is requested, in order to

ensure that the COMMAREA passed to the linked program is of sufficient size.

The initial value of this field MQCODL_AS_INPUT.

FacilityKeepTime (MQLONG)
Bridge facility release time.

This is a reserved field. The value must be 0.

ADSDescriptor (MQLONG)
Send/receive ADS descriptor.

This is a reserved field. The value must be 0.

ConversationalTask (MQLONG)
Whether task can be conversational.

This is a reserved field. The value must be 0.

TaskEndStatus (MQLONG)
Status at end of task.

This is a reserved field. The value must be 0.

Facility (MQBYTE8)
BVT token value.

This is a reserved field. The value must be 8 nulls. The length of this field is given by MQ_FACILITY_LENGTH.

Function (MQCHAR4)
MQ call name or CICS EIBFN function.

The value returned in this field is dependent on *ReturnCode*; see Table 20 on page 22. The following values are possible when *Function* contains an MQ call name:

MQCFUNC_MQCONN
MQCONN call.

MQCFUNC_MQGET
MQGET call.

MQCFUNC_MQINQ
MQINQ call.

MQCFUNC_MQOPEN
MQOPEN call.

MQCFUNC_MQPUT
MQPUT call.

MQCFUNC_MQPUT1
MQPUT1 call.

MQCFUNC_NONE
No call.

In all cases, for the C programming language the constants MQCFUNC_*_ARRAY are also defined; these have the same values as the corresponding MQCFUNC_* constants, but are arrays of characters instead of strings.

The length of this field is given by MQ_FUNCTION_LENGTH. The initial value of this field is MQCFUNC_NONE.

AbendCode (MQCHAR4)
Abend code.

The value returned in this field is dependent on *ReturnCode*; see Table 20 on page 22.

The length of this field is given by MQ_ABEND_CODE_LENGTH. The initial value of this field is 4 blank characters.

Authenticator (MQCHAR8)
Password or passticket.

This is a password or passticket. If user-identifier authentication is active for the CICS bridge, *Authenticator* is used with the user identifier in the MQMD identity context to authenticate the sender of the message.

The length of this field is given by MQ_AUTHENTICATOR_LENGTH. The initial value of this field is 8 blank characters.

Reserved1 (MQCHAR8)
Reserved.

This is a reserved field. The value must be 8 blanks.

ReplyToFormat (MQCHAR8)
MQ format name of reply message.

This is the MQ format name of the reply message which will be sent in response to the current message. The rules for coding this are the same as those for the *Format* field in MQMD.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

RemoteSysId (MQCHAR4)
Remote sysid to use.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_REMOTE_SYS_ID_LENGTH.

RemoteTransId (MQCHAR4)
Remote transid to attach.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_TRANSACTION_ID_LENGTH.

TransactionId (MQCHAR4)
Transaction to attach.

This field is applicable only when *UOWControl* has the value MQCUOWC_FIRST or MQCUOWC_ONLY. *TransactionId* is the transaction code under which all programs within the unit of work are to be run. If the value specified is blank, the CICS bridge default transaction code is used.

The initial value of this field is 4 blanks. The length of this field is given by MQ_TRANSACTION_ID_LENGTH.

FacilityLike (MQCHAR4)
Terminal emulated attributes.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_FACILITY_LIKE_LENGTH.

AttentionId (MQCHAR4)
AID key.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_ATTENTION_ID_LENGTH.

StartCode (MQCHAR4)
Transaction start code.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_START_CODE_LENGTH.

CancelCode (MQCHAR4)
Abend transaction code.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_CANCEL_CODE_LENGTH.

NextTransactionId (MQCHAR4)
Next transaction to attach.

This is a reserved field. The value must be 4 blanks. The length of this field is given by MQ_TRANSACTION_ID_LENGTH.

Reserved2 (MQCHAR8)
Reserved.

This is a reserved field. The value must be 8 blanks.

Reserved3 (MQCHAR8)
Reserved.

This is a reserved field. The value must be 8 blanks.

MQFB_★ feedback codes

The following new MQ feedback codes are used by the CICS bridge:

MQFB_CICS_APPL_ABENDED
Application abended.

The application program specified in the message abended. This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_APPL_NOT_STARTED
Application cannot be started.

The EXEC CICS LINK for the application program specified in the message failed. This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_BRIDGE_FAILURE
CICS bridge terminated abnormally without completing normal error processing.

MQFB_CICS_CCSID_ERROR
Character set identifier not valid.

MQFB_CICS_CIH_ERROR
CICS information header structure missing or not valid.

MQFB_CICS_COMMAREA_ERROR
Length of CICS commarea not valid.

MQFB_CICS_CORREL_ID_ERROR
Correlation identifier not valid.

MQFB_CICS_DLQ_ERROR
Dead-letter queue not available.

The CICS bridge task was unable to copy a reply to this request to the dead-letter queue. The request was backed out.

MQFB_CICS_ENCODING_ERROR
Encoding not valid.

MQFB_CICS_INTERNAL_ERROR
CICS bridge encountered unexpected error.

This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_NOT_AUTHORIZED
User identifier not authorized or password not valid.

This feedback code occurs only in the *Reason* field of the MQDLH structure.

MQFB_CICS_UOW_BACKED_OUT
Unit of work backed out.

The unit of work was backed out, for one of the following reasons:

- A failure was detected whilst processing another request within the same unit of work.
- A CICS abend occurred whilst the unit of work was in progress.

MQFB_CICS_UOW_ERROR
Unit-of-work control field *UOWControl* not valid.

Table 21 (Page 1 of 2). Initial values of fields in MQCIH

Field name	Name of constant	Value of constant
StrucId	MQCIH_STRUC_ID	'CIHb' (See note 1)
Version	MQCIH_VERSION_1	1
StrucLength	MQCIH_LENGTH_1	164
Encoding	None	0
CodedCharSetId	None	0
Format	MQFMT_NONE	'bbbbbbbb'
Flags	MQCIH_NONE	0
ReturnCode	MQCRC_OK	0
CompCode	MQCC_OK	0
Reason	MQRC_NONE	0
UOWControl	MQCUOWC_ONLY	273

Table 21 (Page 2 of 2). Initial values of fields in MQCIH

Field name	Name of constant	Value of constant
GetWaitInterval	MQCGWI_DEFAULT	-2
LinkType	MQCLT_PROGRAM	1
OutputDataLength	MQCODL_AS_INPUT	-1
FacilityKeepTime	None	0
ADSDescriptor	MQCADSD_NONE	0
ConversationalTask	MQCCT_NO	0
TaskEndStatus	MQCTES_NOSYNC	0
Facility	MQCFAC_NONE	Nulls
Function	MQCFUC_NONE	'bbbb'
AbendCode	None	'bbbb'
Authenticator	None	'bbbbbbbb'
Reserved1	None	'bbbbbbbb'
ReplyToFormat	MQFMT_NONE	'bbbbbbbb'
RemoteSysId	None	'bbbb'
RemoteTransId	None	'bbbb'
TransactionId	None	'bbbb'
FacilityLike	None	'bbbb'
AttentionId	None	'bbbb'
StartCode	MQCSC_NONE	'bbbb'
CancelCode	None	'bbbb'
NextTransactionId	None	'bbbb'
Reserved2	None	'bbbbbbbb'
Reserved3	None	'bbbbbbbb'

Notes:

1. The symbol 'b' represents a single blank character.
2. In the C programming language, the macro variable MQCIH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQCIH MyCIH = {MQCIH_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQCIH {
  MQCHAR4  StrucId;            /* Structure identifier */
  MQLONG   Version;            /* Structure version number */
  MQLONG   StrucLength;        /* Length of MQCIH structure */
  MQLONG   Encoding;           /* Reserved */
  MQLONG   CodedCharSetId;     /* Reserved */
  MQCHAR8  Format;             /* MQ format name */
  MQLONG   Flags;              /* Reserved */
  MQLONG   ReturnCode;         /* Return code from bridge */
  MQLONG   CompCode;           /* MQ completion code or CICS EIBRESP */
  MQLONG   Reason;             /* MQ reason or feedback code, or CICS
                                  EIBRESP2 */
```

```
  MQLONG   UOWControl;          /* Unit-of-work control */
  MQLONG   GetWaitInterval;     /* Wait interval for MQGET call issued
                                   by bridge task */
  MQLONG   LinkType;            /* Link type */
  MQLONG   OutputDataLength;    /* Output COMMAREA data length */
  MQLONG   FacilityKeepTime;    /* Bridge facility release time */
  MQLONG   ADSDescriptor;       /* Send/receive ADS descriptor */
  MQLONG   ConversationalTask;  /* Whether task can be conversational */
  MQLONG   TaskEndStatus;       /* Status at end of task */
  MQBYTE8  Facility;            /* BVT token value */
  MQCHAR4  Function;            /* MQ call name or CICS EIBFN
                                   function */
  MQCHAR4  AbendCode;           /* Abend code */
  MQCHAR8  Authenticator;       /* Password or passticket */
  MQCHAR8  Reserved1;           /* Reserved */
  MQCHAR8  ReplyToFormat;       /* MQ format name of reply message */
  MQCHAR4  RemoteSysId;         /* Remote sysid to use */
  MQCHAR4  RemoteTransId;       /* Remote transid to attach */
  MQCHAR4  TransactionId;       /* Transaction to attach */
  MQCHAR4  FacilityLike;        /* Terminal emulated attributes */
  MQCHAR4  AttentionId;         /* AID key */
  MQCHAR4  StartCode;           /* Transaction start code */
  MQCHAR4  CancelCode;          /* Abend transaction code */
  MQCHAR4  NextTransactionId;   /* Next transaction to attach */
  MQCHAR8  Reserved2;           /* Reserved */
  MQCHAR8  Reserved3;           /* Reserved */
 } MQCIH;
```

COBOL language declaration

```
**   MQCIH structure
  10 MQCIH.
**    Structure identifier
    15 MQCIH-STRUCID            PIC X(4).
**    Structure version number
    15 MQCIH-VERSION            PIC S9(9) BINARY.
**    Length of MQCIH structure
    15 MQCIH-STRUCLENGTH        PIC S9(9) BINARY.
**    Reserved
    15 MQCIH-ENCODING           PIC S9(9) BINARY.
**    Reserved
    15 MQCIH-CODEDCHARSETID     PIC S9(9) BINARY.
**    MQ format name
    15 MQCIH-FORMAT             PIC X(8).
**    Reserved
    15 MQCIH-FLAGS              PIC S9(9) BINARY.
**    Return code from bridge
    15 MQCIH-RETURNCODE         PIC S9(9) BINARY.
**    MQ completion code or CICS EIBRESP
    15 MQCIH-COMPCODE           PIC S9(9) BINARY.
**    MQ reason or feedback code, or CICS EIBRESP2
    15 MQCIH-REASON             PIC S9(9) BINARY.
**    Unit-of-work control
    15 MQCIH-UOWCONTROL         PIC S9(9) BINARY.
**    Wait interval for MQGET call issued by bridge task
    15 MQCIH-GETWAITINTERVAL    PIC S9(9) BINARY.
**    Link type
```

```
                    15 MQCIH-LINKTYPE           PIC S9(9) BINARY.
                  **    Output COMMAREA data length
                    15 MQCIH-OUTPUTDATALENGTH   PIC S9(9) BINARY.
                  **    Bridge facility release time
                    15 MQCIH-FACILITYKEEPTIME   PIC S9(9) BINARY.
                  **    Send/receive ADS descriptor
                    15 MQCIH-ADSDESCRIPTOR      PIC S9(9) BINARY.
                  **    Whether task can be conversational
                    15 MQCIH-CONVERSATIONALTASK PIC S9(9) BINARY.
                  **    Status at end of task
                    15 MQCIH-TASKENDSTATUS      PIC S9(9) BINARY.
                  **    BVT token value
                    15 MQCIH-FACILITY           PIC X(8).
                  **    MQ call name or CICS EIBFN function
                    15 MQCIH-FUNCTION           PIC X(4).
                  **    Abend code
                    15 MQCIH-ABENDCODE          PIC X(4).
                  **    Password or passticket
                    15 MQCIH-AUTHENTICATOR      PIC X(8).
                  **    Reserved
                    15 MQCIH-RESERVED1          PIC X(8).
                  **    MQ format name of reply message
                    15 MQCIH-REPLYTOFORMAT      PIC X(8).
                  **    Remote sysid to use
                    15 MQCIH-REMOTESYSID        PIC X(4).
                  **    Remote transid to attach
                    15 MQCIH-REMOTETRANSID      PIC X(4).
                  **    Transaction to attach
                    15 MQCIH-TRANSACTIONID      PIC X(4).
                  **    Terminal emulated attributes
                    15 MQCIH-FACILITYLIKE       PIC X(4).
                  **    AID key
                    15 MQCIH-ATTENTIONID        PIC X(4).
                  **    Transaction start code
                    15 MQCIH-STARTCODE          PIC X(4).
                  **    Abend transaction code
                    15 MQCIH-CANCELCODE         PIC X(4).
                  **    Next transaction to attach
                    15 MQCIH-NEXTTRANSACTIONID  PIC X(4).
                  **    Reserved
                    15 MQCIH-RESERVED2          PIC X(8).
                  **    Reserved
                    15 MQCIH-RESERVED3          PIC X(8).
```

PL/I language declaration

```
dcl
 1 MQCIH based,
  3 StrucId            char(4),       /* Structure identifier */
  3 Version            fixed bin(31), /* Structure version number */
  3 StrucLength        fixed bin(31), /* Length of MQCIH structure */
  3 Encoding           fixed bin(31), /* Reserved */
  3 CodedCharSetId     fixed bin(31), /* Reserved */
  3 Format             char(8),       /* MQ format name */
  3 Flags              fixed bin(31), /* Reserved */
  3 ReturnCode         fixed bin(31), /* Return code from bridge */
  3 CompCode           fixed bin(31), /* MQ completion code or CICS
                                         EIBRESP */
```

```
  3 Reason             fixed bin(31), /* MQ reason or feedback code, or
                                         CICS EIBRESP2 */
  3 UOWControl         fixed bin(31), /* Unit-of-work control */
  3 GetWaitInterval    fixed bin(31), /* Wait interval for MQGET call
                                         issued by bridge task */
  3 LinkType           fixed bin(31), /* Link type */
  3 OutputDataLength   fixed bin(31), /* Output COMMAREA data length */
  3 FacilityKeepTime   fixed bin(31), /* Bridge facility release time */
  3 ADSDescriptor      fixed bin(31), /* Send/receive ADS descriptor */
  3 ConversationalTask fixed bin(31), /* Whether task can be conversa-
                                         tional */
  3 TaskEndStatus      fixed bin(31), /* Status at end of task */
  3 Facility           char(8),       /* BVT token value */
  3 Function           char(4),       /* MQ call name or CICS EIBFN
                                         function */
  3 AbendCode          char(4),       /* Abend code */
  3 Authenticator      char(8),       /* Password or passticket */
  3 Reserved1          char(8),       /* Reserved */
  3 ReplyToFormat      char(8),       /* MQ format name of reply
                                         message */
  3 RemoteSysId        char(4),       /* Remote sysid to use */
  3 RemoteTransId      char(4),       /* Remote transid to attach */
  3 TransactionId      char(4),       /* Transaction to attach */
  3 FacilityLike       char(4),       /* Terminal emulated attributes */
  3 AttentionId        char(4),       /* AID key */
  3 StartCode          char(4),       /* Transaction start code */
  3 CancelCode         char(4),       /* Abend transaction code */
  3 NextTransactionId  char(4),       /* Next transaction to attach */
  3 Reserved2          char(8),       /* Reserved */
  3 Reserved3          char(8);       /* Reserved */
```

System/390 assembler language declaration

```
MQCIH                    DSECT
MQCIH_STRUCID            DS   CL4      Structure identifier
MQCIH_VERSION            DS   F        Structure version number
MQCIH_STRUCLENGTH        DS   F        Length of MQCIH structure
MQCIH_ENCODING           DS   F        Reserved
MQCIH_CODEDCHARSETID     DS   F        Reserved
MQCIH_FORMAT             DS   CL8      MQ format name
MQCIH_FLAGS              DS   F        Reserved
MQCIH_RETURNCODE         DS   F        Return code from bridge
MQCIH_COMPCODE           DS   F        MQ completion code or CICS
*                                      EIBRESP
MQCIH_REASON             DS   F        MQ reason or feedback code,
*                                      or CICS EIBRESP2
MQCIH_UOWCONTROL         DS   F        Unit-of-work control
MQCIH_GETWAITINTERVAL    DS   F        Wait interval for MQGET call
*                                      issued by bridge task
MQCIH_LINKTYPE           DS   F        Link type
MQCIH_OUTPUTDATALENGTH   DS   F        Output COMMAREA data length
MQCIH_FACILITYKEEPTIME   DS   F        Bridge facility release time
MQCIH_ADSDESCRIPTOR      DS   F        Send/receive ADS descriptor
MQCIH_CONVERSATIONALTASK DS   F        Whether task can be
*                                      conversational
MQCIH_TASKENDSTATUS      DS   F        Status at end of task
MQCIH_FACILITY           DS   XL8      BVT token value
MQCIH_FUNCTION           DS   CL4      MQ call name or CICS EIBFN
```

```
*                                                  function
MQCIH_ABENDCODE                DS   CL4      Abend code
MQCIH_AUTHENTICATOR            DS   CL8      Password or passticket
MQCIH_RESERVED1                DS   CL8      Reserved
MQCIH_REPLYTOFORMAT            DS   CL8      MQ format name of reply
*                                            message
MQCIH_REMOTESYSID              DS   CL4      Remote sysid to use
MQCIH_REMOTETRANSID            DS   CL4      Remote transid to attach
MQCIH_TRANSACTIONID            DS   CL4      Transaction to attach
MQCIH_FACILITYLIKE             DS   CL4      Terminal emulated attributes
MQCIH_ATTENTIONID              DS   CL4      AID key
MQCIH_STARTCODE                DS   CL4      Transaction start code
MQCIH_CANCELCODE               DS   CL4      Abend transaction code
MQCIH_NEXTTRANSACTIONID        DS   CL4      Next transaction to attach
MQCIH_RESERVED2                DS   CL8      Reserved
MQCIH_RESERVED3                DS   CL8      Reserved
MQCIH_LENGTH                   EQU  *-MQCIH  Length of structure
                               ORG  MQCIH
MQCIH_AREA                     DS   CL(MQCIH_LENGTH)
```

MQCNO – Connect options

The following table summarizes the fields in the structure.

Table 22. Fields in MQCNO

Field	Description	Page
StrucId	Structure identifier	35
Version	Structure version number	35
Options	Options that control the action of MQCONNX	35

The MQCNO structure is an input/output parameter for the MQCONNX call.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, Sun Solaris, Windows client, Windows NT.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQCNO_STRUC_ID
Identifier for connect-options structure.

For the C programming language, the constant MQCNO_STRUC_ID_ARRAY is also defined; this has the same value as MQCNO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQCNO_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQCNO_VERSION_1
Version number for connect-options structure.

The following constant specifies the version number of the current version:

MQCNO_CURRENT_VERSION
Current version of connect-options structure.

This is always an input field. The initial value of this field is MQCNO_VERSION_1.

Options (MQLONG)
Options that control the action of MQCONNX.

The following options control the type of MQ binding that will be used; only one of these options can be specified:

MQCNO_STANDARD_BINDING
Standard binding.

This option causes the application and the local-queue-manager

agent (the component that manages queuing operations) to run in separate units of execution (generally, in separate processes). This arrangement maintains the integrity of the queue manager, that is, it protects the queue manager from errant programs.

MQCNO_STANDARD_BINDING should be used in situations where the application may not have been fully tested, or may be unreliable or untrustworthy. MQCNO_STANDARD_BINDING is the default.

MQCNO_STANDARD_BINDING is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

MQCNO_FASTPATH_BINDING
Fast-path binding.

This option causes the application and the local-queue-manager agent to be part of the same unit of execution. This is in contrast to the normal method of binding, where the application and the local-queue-manager agent run in separate units of execution.

MQCNO_FASTPATH_BINDING is ignored if specified by an MQ client application; processing continues as though the option had not been specified.

MQCNO_FASTPATH_BINDING may be of advantage in situations where the use of multiple processes is a significant performance overhead compared to the overall resource used by the application.

Note: An application that uses the fastpath binding is known as a *trusted application*.

The following important points must be considered when deciding whether to use the fast-path binding:

1. **Use of the MQCNO_FASTPATH_BINDING option compromises the integrity of the queue manager, as it permits a rogue application to alter or corrupt messages and other data areas belonging to the queue manager. It should therefore be considered for use *only* in situations where these issues have been fully evaluated.**
2. On Windows NT, use of MQCNO_FASTPATH_BINDING requires that the program be a member of the `mqm` group.
3. On UNIX systems, use of MQCNO_FASTPATH_BINDING requires that the program run with the `mqm` user identifier and the `mqm` group identifier. The application can be made to run this way by configuring the program so that it is owned by the `mqm` user identifier and `mqm` group identifier, and then setting the `setuid` and `setgid` permission bits on the program.
4. On OS/2, UNIX systems, and Windows NT, a program that uses MQCNO_FASTPATH_BINDING cannot have more than one thread connected to a queue manager at any one time.
5. You must not use asynchronous signals and timer interrupts (such as `sigkill`) with MQCNO_FASTPATH_BINDING. There are also restrictions on the use of shared memory segments.

6. You must explicitly disconnect trusted applications from the queue manager.
7. You must stop trusted applications before ending the queue manager with the `endmqm` command.

For more information about the implications of using trusted applications, see the *MQSeries Application Programming Guide*.

On AIX, HP-UX, OS/2, Sun Solaris, and Windows NT, the environment variable `MQ_CONNECT_TYPE` can be used in association with the bind type specified by the *Options* field, to control the type of binding used. If this environment variable is specified, it should have the value `FASTPATH` or `STANDARD`; if it has some other value, it is ignored. The value of the environment variable is case sensitive.

The environment variable and *Options* field interact as follows:

- If the environment variable is not specified, or has a value which is not supported, use of the fast-path binding is determined solely by the *Options* field.
- If the environment variable is specified and has a supported value, the fast-path binding is used only if *both* the environment variable and *Options* field specify the fast-path binding.

If none of the options described above is specified, the following option can be used:

MQCNO_NONE
No options specified.

MQCNO_NONE is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is always an input field. The initial value of this field is MQCNO_NONE.

Table 23. Initial values of fields in MQCNO

Field name	Name of constant	Value of constant
StrucId	MQCNO_STRUC_ID	`'CNOb'` (See note 1)
Version	MQCNO_VERSION_1	`1`
Options	MQCNO_NONE	`0`

Notes:

1. The symbol 'b' represents a single blank character.
2. In the C programming language, the macro variable MQCNO_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQCNO MyCNO = {MQCNO_DEFAULT};
```

C language declaration

```
typedef struct tagMQCNO {
  MQCHAR4  StrucId;  /* Structure identifier */
  MQLONG   Version;  /* Structure version number */
  MQLONG   Options;  /* Options that control the action of MQCONNX */
 } MQCNO;
```

COBOL language declaration

```
**   MQCNO structure
  10 MQCNO.
**    Structure identifier
   15 MQCNO-STRUCID PIC X(4).
**    Structure version number
   15 MQCNO-VERSION PIC S9(9) BINARY.
**    Options that control the action of MQCONNX
   15 MQCNO-OPTIONS PIC S9(9) BINARY.
```

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

```
dcl
 1 MQCNO based,
  3 StrucId char(4),       /* Structure identifier */
  3 Version fixed bin(31), /* Structure version number */
  3 Options fixed bin(31); /* Options that control the action of
                              MQCONNX */
```

MQDH – Distribution header

The following table summarizes the fields in the structure.

Table 24. Fields in MQDH

Field	Description	Page
StrucId	Structure identifier	40
Version	Structure version number	40
StrucLength	Length of MQDH structure plus following records	40
Encoding	Encoding of message data	41
CodedCharSetId	Coded character-set identifier of message data	41
Format	Format name of message data	41
Flags	General flags	41
PutMsgRecFields	Flags indicating which MQPMR fields are present	42
RecsPresent	Number of object records present	42
ObjectRecOffset	Offset of first object record from start of MQDH	42
PutMsgRecOffset	Offset of first put-message record from start of MQDH	42

The MQDH structure describes the data that is present in a message on a transmission queue when that message is a distribution-list message (that is, the message is being sent to multiple destination queues). This structure is for use by specialized applications that put messages directly on transmission queues, or which remove messages from transmission queues (for example: message channel agents).

This structure should *not* be used by normal applications which simply want to put messages to distribution lists. Those applications should use the MQOD structure to define the destinations in the distribution list, and the MQPMO structure to specify message properties or receive information about the messages sent to the individual destinations.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

When an application puts a message to a distribution list, and some or all of the destinations are remote, the queue manager prefixes the application message data with the MQXQH and MQDH structures, and places the message on the relevant transmission queue. The data therefore occurs in the following sequence when the message is on a transmission queue:

- MQXQH structure
- MQDH structure
- Application message data

Depending on the destinations, more than one such message may be generated by the queue manager, and placed on different transmission queues. In this case, the MQDH structures in those messages identify different subsets of the destinations defined by the distribution list opened by the application.

An application that puts a distribution-list message directly on a transmission queue must conform to the sequence described above, and must ensure that the MQDH structure is correct. If the MQDH structure is not valid, the queue manager may choose to fail the MQPUT or MQPUT1 call with reason code MQRC_DH_ERROR.

Messages can be stored on a queue in distribution-list form only if the queue is defined as being able to support distribution list messages (see the *DistLists* queue attribute described in “Attributes for local queues and model queues” on page 348). If an application puts a distribution-list message directly on a queue that does not support distribution lists, the queue manager splits the distribution list message into individual messages, and places those on the queue instead.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQDH_STRUC_ID
Identifier for distribution header structure.

For the C programming language, the constant MQDH_STRUC_ID_ARRAY is also defined; this has the same value as MQDH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQDH_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQDH_VERSION_1
Version number for distribution header structure.

The following constant specifies the version number of the current version:

MQDH_CURRENT_VERSION
Current version of distribution header structure.

The initial value of this field is MQDH_VERSION_1.

StrucLength (MQLONG)
Length of MQDH structure plus following records.

This is the number of bytes from the start of the MQDH structure to the start of the message data following the arrays of MQOR and MQPMR records. The data occurs in the following sequence:

- MQDH structure
- Array of MQOR records
- Array of MQPMR records
- Message data

The arrays of MQOR and MQPMR records are addressed by offsets contained within the MQDH structure. If these offsets result in unused bytes between one or more of the MQDH structure, the arrays of records, and the message data, those unused bytes must be included in the value

of *StrucLength*, but the content of those bytes is not preserved by the queue manager. It is valid for the array of MQPMR records to precede the array of MQOR records.

The initial value of this field is 0.

Encoding (MQLONG)
Encoding of message data.

The initial value of this field is 0.

CodedCharSetId (MQLONG)
Coded character-set identifier of message data.

The initial value of this field is 0.

Format (MQCHAR8)
Format name of message data.

The initial value of this field is MQFMT_NONE.

Flags (MQLONG)
General flags.

The following flag can be specified:

MQDHF_NEW_MSG_IDS
Generate new message identifiers.

This flag indicates that a new message identifier is to be generated for each destination in the distribution list. This can be set only when there are no put-message records present, or when the records are present but they do not contain the *MsgId* field.

Using this flag defers generation of the message identifiers until the last possible moment, namely the moment when the distribution-list message is finally split into individual messages. This minimizes the amount of control information that must flow with the distribution-list message.

When an application puts a message to a distribution list, the queue manager sets MQDHF_NEW_MSG_IDS in the MQDH it generates when both of the following are true:

- There are no put-message records provided by the application, or the records provided do not contain the *MsgId* field.
- The *MsgId* field in MQMD is MQMI_NONE, or the *Options* field in MQPMO includes MQPMO_NEW_MSG_ID

If no flags are needed, the following can be specified:

MQDHF_NONE
No flags.

This constant indicates that no flags have been specified. MQDHF_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is MQDHF_NONE.

PutMsgRecFields (MQLONG)
Flags indicating which MQPMR fields are present.

Zero or more of the following flags can be specified:

MQPMRF_MSG_ID
Message-identifier field is present.

MQPMRF_CORREL_ID
Correlation-identifier field is present.

MQPMRF_GROUP_ID
Group-identifier field is present.

MQPMRF_FEEDBACK
Feedback field is present.

MQPMRF_ACCOUNTING_TOKEN
Accounting-token field is present.

If no MQPMR fields are present, the following can be specified:

MQPMRF_NONE
No put-message record fields are present.

MQPMRF_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is MQPMRF_NONE.

RecsPresent (MQLONG)
Number of object records present.

This defines the number of destinations. A distribution list must always contain at least one destination, so *RecsPresent* must always be greater than zero.

The initial value of this field is 0.

ObjectRecOffset (MQLONG)
Offset of first object record from start of MQDH.

This field gives the offset in bytes of the first record in the array of MQOR object records containing the names of the destination queues. There are *RecsPresent* records in this array. These records (plus any bytes skipped between the first object record and the previous field) are included in the length given by the *StrucLength* field.

A distribution list must always contain at least one destination, so *ObjectRecOffset* must always be greater than zero.

The initial value of this field is 0.

PutMsgRecOffset (MQLONG)
Offset of first put message record from start of MQDH.

This field gives the offset in bytes of the first record in the array of MQPMR put message records containing the message properties. If present, there are *RecsPresent* records in this array. These records (plus any bytes skipped between the first put message record and the previous field) are included in the length given by the *StrucLength* field.

Put message records are optional; if no records are provided, *PutMsgRecOffset* is zero, and *PutMsgRecFields* has the value MQPMRF_NONE.

The initial value of this field is 0.

Table 25. Initial values of fields in MQDH

Field name	Name of constant	Value of constant
StrucId	MQDH_STRUC_ID	'DHbb' (See note 1)
Version	MQDH_VERSION_1	1
StrucLength	None	0
Encoding	None	0
CodedCharSetId	None	0
Format	MQFMT_NONE	'bbbbbbbb'
Flags	MQDHF_NONE	0
PutMsgRecFields	MQPMRF_NONE	0
RecsPresent	None	0
ObjectRecOffset	None	0
PutMsgRecOffset	None	0

Notes:

1. The symbol 'b' represents a single blank character.
2. In the C programming language, the macro variable MQDH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQDH MyDH = {MQDH_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQDH {
  MQCHAR4  StrucId;          /* Structure identifier */
  MQLONG   Version;          /* Structure version number */
  MQLONG   StrucLength;      /* Length of MQDH structure plus following
                                records */
  MQLONG   Encoding;         /* Encoding of message data */
  MQLONG   CodedCharSetId;   /* Coded character-set identifier of
                                message data */
  MQCHAR8  Format;           /* Format name of message data */
  MQLONG   Flags;            /* General flags */
  MQLONG   PutMsgRecFields;  /* Flags indicating which MQPMR fields are
                                present */
  MQLONG   RecsPresent;      /* Number of object records present */
  MQLONG   ObjectRecOffset;  /* Offset of first object record from start
                                of MQDH */
  MQLONG   PutMsgRecOffset;  /* Offset of first put message record from
                                start of MQDH */
 } MQDH;
```

COBOL language declaration

```
**   MQDH structure
  10 MQDH.
**    Structure identifier
   15 MQDH-STRUCID         PIC X(4).
**    Structure version number
   15 MQDH-VERSION         PIC S9(9) BINARY.
**    Length of MQDH structure plus following records
   15 MQDH-STRUCLENGTH     PIC S9(9) BINARY.
**    Encoding of message data
   15 MQDH-ENCODING        PIC S9(9) BINARY.
**    Coded character-set identifier of message data
   15 MQDH-CODEDCHARSETID  PIC S9(9) BINARY.
**    Format name of message data
   15 MQDH-FORMAT          PIC X(8).
**    General flags
   15 MQDH-FLAGS           PIC S9(9) BINARY.
**    Flags indicating which MQPMR fields are present
   15 MQDH-PUTMSGRECFIELDS PIC S9(9) BINARY.
**    Number of object records present
   15 MQDH-RECSPRESENT     PIC S9(9) BINARY.
**    Offset of first object record from start of MQDH
   15 MQDH-OBJECTRECOFFSET PIC S9(9) BINARY.
**    Offset of first put message record from start of MQDH
   15 MQDH-PUTMSGRECOFFSET PIC S9(9) BINARY.
```

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

```
dcl
 1 MQDH based,
  3 StrucId         char(4),       /* Structure identifier */
  3 Version         fixed bin(31), /* Structure version number */
  3 StrucLength     fixed bin(31), /* Length of MQDH structure plus fol-
                                      lowing records */
  3 Encoding        fixed bin(31), /* Encoding of message data */
  3 CodedCharSetId  fixed bin(31), /* Coded character-set identifier of
                                      message data */
  3 Format          char(8),       /* Format name of message data */
  3 Flags           fixed bin(31), /* General flags */
  3 PutMsgRecFields fixed bin(31), /* Flags indicating which MQPMR
                                      fields are present */
  3 RecsPresent     fixed bin(31), /* Number of object records
                                      present */
  3 ObjectRecOffset fixed bin(31), /* Offset of first object record from
                                      start of MQDH */
  3 PutMsgRecOffset fixed bin(31); /* Offset of first put message record
                                      from start of MQDH */
```

MQDLH – Dead-letter header

The following table summarizes the fields in the structure.

Table 26. Fields in MQDLH

Field	Description	Page
StrucId	Structure identifier	47
Version	Structure version number	47
Reason	Reason message arrived on dead-letter queue	47
DestQName	Name of original destination queue	48
DestQMgrName	Name of original destination queue manager	49
Encoding	Original data encoding	49
CodedCharSetId	Original coded character set identifier	49
Format	Original format name	49
PutApplType	Type of application that put message on dead-letter queue	50
PutApplName	Name of application that put message on dead-letter queue	50
PutDate	Date when message was put on dead-letter queue	50
PutTime	Time when message was put on dead-letter queue	51

The MQDLH structure describes the information that is prefixed to the application message data of messages on the dead-letter (undelivered-message) queue. A message can arrive on the dead-letter queue either because the queue manager or message channel agent has redirected it to the queue, or because an application has put the message directly on the queue.

Special processing is done when a message which is a segment is put with an MQDLH structure at the front; see the description of the MQMDE structure for further details.

This structure is *not* supported in the following environments: 16-bit Windows, 32-bit Windows.

Applications that 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 should consider doing one of the following:

- Truncate the message data to fit on the dead-letter queue.
- Record the message on auxiliary storage and place an exception report message on the dead-letter queue indicating this.
- Discard the message and return an error to its originator. If the message is (or might be) a critical message, this should be done only if it is known that the

originator still has a copy of the message—for example, a message received by a message channel agent from a communication channel.

Which of the above is appropriate (if any) depends on the design of the application.

When a message is put on the dead-letter queue, all of the fields in the message descriptor MQMD should be copied from those in the original message descriptor (if there is one), with the exception of the following:

- The *CodedCharSetId* and *Encoding* fields should be set to whatever character set and encoding are used for fields in the MQDLH structure.
- The *Format* field should be set to MQFMT_DEAD_LETTER_HEADER to indicate that the data begins with a MQDLH structure.
- The context fields:

  UserIdentifier
  AccountingToken
  ApplIdentityData
  PutApplType
  PutApplName
  PutDate
  PutTime
  ApplOriginData

  should be set by using a context option appropriate to the nature of the program:

  – A program putting on the dead-letter queue a message that is not related to any preceding message should use the MQPMO_DEFAULT_CONTEXT option; this causes the queue manager to set all of the context fields in the message descriptor to their default values.

  – A program putting on the dead-letter queue a message it has just received should use the MQPMO_PASS_ALL_CONTEXT option, in order to preserve the original context information.

  – A program putting on the dead-letter queue a *reply* to a message it has just received should use the MQPMO_PASS_IDENTITY_CONTEXT option; this preserves the identity information but sets the origin information to be that of the server.

  – A message channel agent putting on the dead-letter queue a message it received from its communication channel should use the MQPMO_SET_ALL_CONTEXT option, to preserve the original context information.

In the MQDLH structure itself, the fields should be set as follows:

- The *CodedCharSetId*, *Encoding* and *Format* fields should be set to the values that describe the application message data that follows the MQDLH structure—usually the values from the original message descriptor.
- The context fields *PutApplType*, *PutApplName*, *PutDate*, and *PutTime* should be set to values appropriate to the application that is putting the message on the dead-letter queue; these values are not related to the original message.
- Other fields should be set as appropriate.

Character data in the MQDLH structure should be in the character set defined by the *CodedCharSetId* field of the message descriptor. Numeric data in the MQDLH structure should be in the data encoding defined by the *Encoding* field of the message descriptor. The application should ensure that all fields have valid values, and that character fields are padded with blanks to the defined length of the field; the character data should not be terminated prematurely by using a null character, because the queue manager does not convert the null and subsequent characters to blanks in the MQDLH structure.

Applications that get messages from the dead-letter queue should verify that the messages begin with an MQDLH structure. The application can determine whether an MQDLH structure is present by examining the *Format* field in the message descriptor MQMD; if the field has the value MQFMT_DEAD_LETTER_HEADER, the message data begins with an MQDLH structure. Applications that get messages from the dead-letter queue should also be aware that such messages may have been truncated if they were originally too long for the queue.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQDLH_STRUC_ID
Identifier for dead-letter header structure.

For the C programming language, the constant MQDLH_STRUC_ID_ARRAY is also defined; this has the same value as MQDLH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQDLH_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQDLH_VERSION_1
Version number for dead-letter header structure.

The following constant specifies the version number of the current version:

MQDLH_CURRENT_VERSION
Current version of dead-letter header structure.

The initial value of this field is MQDLH_VERSION_1.

Reason (MQLONG)
Reason message arrived on dead-letter (undelivered-message) queue.

This identifies the reason why the message was placed on the dead-letter queue instead of on the original destination queue. It should be one of the MQFB_★ or MQRC_★ values (for example, MQRC_Q_FULL). See the description of the *Feedback* field in “MQMD – Message descriptor” on page 98 for details of the common MQFB_★ values that can occur.

If the value is in the range MQFB_IMS_FIRST through MQFB_IMS_LAST, the actual IMS error code can be determined by subtracting MQFB_IMS_ERROR from the value of the *Reason* field.

Some MQFB_★ values only ever occur in this field. They relate to trigger or transmission-queue messages that have been transferred to the dead-letter queue. These are:

MQFB_APPL_CANNOT_BE_STARTED
: Application cannot be started.

  An application processing a trigger message was unable to start the application named in the *ApplId* field of the trigger message (see "MQTM – Trigger message" on page 209).

  On MVS/ESA, the CKTI CICS transaction is an example of an application that processes trigger messages.

MQFB_TM_ERROR
: MQTM structure not valid or missing.

  The *Format* field in MQMD specifies MQFMT_TRIGGER, but the message does not begin with a valid MQTM structure. For example, the *StrucId* mnemonic eye-catcher may not be valid, the *Version* may not be recognized, or the length of the trigger message may be insufficient to contain the MQTM structure.

  On MVS/ESA, the CKTI CICS transaction is an example of an application that processes trigger messages and can generate this feedback code.

MQFB_APPL_TYPE_ERROR
: Application type error.

  An application processing a trigger message was unable to start the application because the *ApplType* field of the trigger message is not valid (see "MQTM – Trigger message" on page 209).

  On MVS/ESA, the CKTI CICS transaction is an example of an application that processes trigger messages.

MQFB_XMIT_Q_MSG_ERROR
: Message on transmission queue not in correct format.

  A message channel agent has found that a message on the transmission queue is not in the correct format. The message channel agent puts the message on the dead-letter queue using this feedback code.

The initial value of this field is MQRC_NONE.

DestQName (MQCHAR48)
: Name of original destination queue.

  This is the name of the message queue that was the original destination for the message.

  The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

DestQMgrName (MQCHAR48)
Name of original destination queue manager.

This is the name of the queue manager that was the original destination for the message.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

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.

The initial value of this field is 0.

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.

The initial value of this field is 0.

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.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

PutApplType (MQLONG)
Type of application that put message on dead-letter (undelivered-message) queue.

This field has the same meaning as the *PutApplType* field in the message descriptor MQMD (see "MQMD – Message descriptor" on page 98 for details).

If it is the queue manager that redirects the message to the dead-letter queue, *PutApplType* has the value MQAT_QMGR.

The initial value of this field is `0`.

PutApplName (MQCHAR28)
Name of application that put message on dead-letter (undelivered-message) queue.

The format of the name depends on the *PutApplType* field. See, also, the description of the *PutApplName* field in "MQMD – Message descriptor" on page 98.

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.

The length of this field is given by MQ_PUT_APPL_NAME_LENGTH. The initial value of this field is the null string in C, and 28 blank characters in other programming languages.

PutDate (MQCHAR8)
Date when message was put on dead-letter (undelivered-message) queue.

The format used for the date when this field is generated by the queue manager is:

YYYYMMDD

where the characters represent:

YYYY	year (four numeric digits)
MM	month of year (01 through 12)
DD	day of month (01 through 31)

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

On OS/2, the queue manager uses the TZ environment variable to calculate GMT. For more information on setting this variable, refer to the *MQSeries System Administration*.

The length of this field is given by MQ_PUT_DATE_LENGTH. The initial value of this field is the null string in C, and 8 blank characters in other programming languages.

PutTime (MQCHAR8)

Time when message was put on the dead-letter (undelivered-message) queue.

The format used for the time when this field is generated by the queue manager is:

HHMMSSTH

where the characters represent (in order):

HH	hours (00 through 23)
MM	minutes (00 through 59)
SS	seconds (00 through 59; see note below)
T	tenths of a second (0 through 9)
H	hundredths of a second (0 through 9)

Note: If the system clock is synchronized to a very accurate time standard, it is possible on rare occasions for 60 or 61 to be returned for the seconds in *PutTime*. This happens when leap seconds are inserted into the global time standard.

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

On OS/2, the queue manager uses the TZ environment variable to calculate GMT. For more information on setting this variable, refer to the *MQSeries System Administration*.

The length of this field is given by MQ_PUT_TIME_LENGTH. The initial value of this field is the null string in C, and 8 blank characters in other programming languages.

Table 27. Initial values of fields in MQDLH

Field name	Name of constant	Value of constant
StrucId	MQDLH_STRUC_ID	'DLHb' (See note 1)
Version	MQDLH_VERSION_1	1
Reason	MQRC_NONE	0
DestQName	None	Blanks (See note 2)
DestQMgrName	None	Blanks
Encoding	None	0
CodedCharSetId	None	0
Format	MQFMT_NONE	'bbbbbbbb'
PutApplType	None	0
PutApplName	None	Blanks
PutDate	None	Blanks
PutTime	None	Blanks

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQDLH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQDLH MyDLH = {MQDLH_DEFAULT} ;
```

C language declaration

```
typedef struct tagMQDLH {
  MQCHAR4   StrucId;         /* Structure identifier */
  MQLONG    Version;         /* Structure version number */
  MQLONG    Reason;          /* Reason message arrived on dead-letter
                                (undelivered-message) queue */
  MQCHAR48  DestQName;       /* Name of original destination queue */
  MQCHAR48  DestQMgrName;    /* Name of original destination queue
                                manager */
  MQLONG    Encoding;        /* Original data encoding */
  MQLONG    CodedCharSetId;  /* Original coded character set
                                identifier */
  MQCHAR8   Format;          /* Original format name */
  MQLONG    PutApplType;     /* Type of application that put message on
                                dead-letter (undelivered-message)
                                queue */
  MQCHAR28  PutApplName;     /* Name of application that put message on
                                dead-letter (undelivered-message)
                                queue */
  MQCHAR8   PutDate;         /* Date when message was put on dead-letter
                                (undelivered-message) queue */
  MQCHAR8   PutTime;         /* Time when message was put on the dead-
                                letter (undelivered-message) queue */
 } MQDLH;
```

COBOL language declaration

```
**   MQDLH structure
  10 MQDLH.
**    Structure identifier
   15 MQDLH-STRUCID        PIC X(4).
**    Structure version number
   15 MQDLH-VERSION        PIC S9(9) BINARY.
**    Reason message arrived on dead-letter (undelivered-message)
**    queue
   15 MQDLH-REASON         PIC S9(9) BINARY.
**    Name of original destination queue
   15 MQDLH-DESTQNAME      PIC X(48).
**    Name of original destination queue manager
   15 MQDLH-DESTQMGRNAME   PIC X(48).
**    Original data encoding
   15 MQDLH-ENCODING       PIC S9(9) BINARY.
**    Original coded character set identifier
   15 MQDLH-CODEDCHARSETID PIC S9(9) BINARY.
**    Original format name
   15 MQDLH-FORMAT         PIC X(8).
**    Type of application that put message on dead-letter
**    (undelivered-message) queue
   15 MQDLH-PUTAPPLTYPE    PIC S9(9) BINARY.
**    Name of application that put message on dead-letter
**    (undelivered-message) queue
   15 MQDLH-PUTAPPLNAME    PIC X(28).
**    Date when message was put on dead-letter
**    (undelivered-message) queue
   15 MQDLH-PUTDATE        PIC X(8).
**    Time when message was put on the dead-letter
**    (undelivered-message) queue
```

```
   15 MQDLH-PUTTIME        PIC X(8).
```

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

```
dcl
 1 MQDLH based,
  3 StrucId        char(4),       /* Structure identifier */
  3 Version        fixed bin(31), /* Structure version number */
  3 Reason         fixed bin(31), /* Reason message arrived on dead-
                                     letter (undelivered-message)
                                     queue */
  3 DestQName      char(48),      /* Name of original destination
                                     queue */
  3 DestQMgrName   char(48),      /* Name of original destination queue
                                     manager */
  3 Encoding       fixed bin(31), /* Original data encoding */
  3 CodedCharSetId fixed bin(31), /* Original coded character set iden-
                                     tifier */
  3 Format         char(8),       /* Original format name */
  3 PutApplType    fixed bin(31), /* Type of application that put
                                     message on dead-letter
                                     (undelivered-message) queue */
  3 PutApplName    char(28),      /* Name of application that put
                                     message on dead-letter
                                     (undelivered-message) queue */
  3 PutDate        char(8),       /* Date when message was put on dead-
                                     letter (undelivered-message)
                                     queue */
  3 PutTime        char(8);       /* Time when message was put on the
                                     dead-letter (undelivered-message)
                                     queue */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQDLH                    DSECT
MQDLH_STRUCID            DS   CL4      Structure identifier
MQDLH_VERSION            DS   F        Structure version number
MQDLH_REASON             DS   F        Reason message arrived on
*                                      dead-letter
*                                      (undelivered-message) queue
MQDLH_DESTQNAME          DS   CL48     Name of original destination
*                                      queue
MQDLH_DESTQMGRNAME       DS   CL48     Name of original destination
*                                      queue manager
MQDLH_ENCODING           DS   F        Original data encoding
MQDLH_CODEDCHARSETID     DS   F        Original coded character set
*                                      identifier
MQDLH_FORMAT             DS   CL8      Original format name
MQDLH_PUTAPPLTYPE        DS   F        Type of application that put
*                                      message on dead-letter
*                                      (undelivered-message) queue
MQDLH_PUTAPPLNAME        DS   CL28     Name of application that put
*                                      message on dead-letter
*                                      (undelivered-message) queue
MQDLH_PUTDATE            DS   CL8      Date when message was put on
*                                      dead-letter
*                                      (undelivered-message) queue
MQDLH_PUTTIME            DS   CL8      Time when message was put on
```

```
*                                                  the dead-letter
*                                                  (undelivered-message) queue
MQDLH_LENGTH                    EQU  *-MQDLH  Length of structure
                                ORG  MQDLH
MQDLH_AREA                      DS   CL(MQDLH_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQDLH^DEF (*);
BEGIN
STRUCT    STRUCID;
BEGIN STRING BYTE [0:3]; END;
INT(32)   VERSION;
INT(32)   REASON;
STRUCT    DESTQNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT    DESTQMGRNAME;
BEGIN STRING BYTE [0:47]; END;
INT(32)   ENCODING;
INT(32)   CODEDCHARSETID;
STRUCT    FORMAT;
BEGIN STRING BYTE [0:7]; END;
INT(32)   PUTAPPLTYPE;
STRUCT    PUTAPPLNAME;
BEGIN STRING BYTE [0:27]; END;
STRUCT    PUTDATE;
BEGIN STRING BYTE [0:7]; END;
STRUCT    PUTTIME;
BEGIN STRING BYTE [0:7]; END;
   END;
```

MQGMO – Get-message options

The following table summarizes the fields in the structure.

Table 28. Fields in MQGMO

Field	Description	Page
StrucId	Structure identifier	56
Version	Structure version number	57
Options	Options that control the action of MQGET	57
WaitInterval	Wait interval	82
Signal1	Signal	82
Signal2	Signal identifier	83
ResolvedQName	Resolved name of destination queue	84
Note: The remaining fields are supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.		
MatchOptions	Options controlling selection criteria used for MQGET	84
GroupStatus	Flag indicating whether message retrieved is in a group	86
SegmentStatus	Flag indicating whether message retrieved is a segment of a logical message	87
Segmentation	Flag indicating whether further segmentation is allowed for the message retrieved	87

The current version of MQGMO is MQGMO_VERSION_2. Fields that exist only in the version-2 structure are identified as such in the descriptions that follow. The declarations of MQGMO provided in the header, COPY, and INCLUDE files for the supported programming languages contain the new fields, but the initial value provided for the *Version* field is MQGMO_VERSION_1; this ensures compatibility with existing applications. To use the new fields, the application must set the version number to MQGMO_VERSION_2. Applications which are intended to be portable between several environments should use a version-2 MQGMO only if all of those environments support version 2.

The version-2 structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

The MQGMO structure is an input/output parameter for the MQGET call.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQGMO_STRUC_ID
Identifier for get-message options structure.

For the C programming language, the constant MQGMO_STRUC_ID_ARRAY is also defined; this has the same

value as MQGMO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQGMO_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be one of the following:

MQGMO_VERSION_1
Version-1 get-message options structure.

This version is supported in all environments.

MQGMO_VERSION_2
Version-2 get-message options structure.

This version is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields that exist only in the version-2 structure are identified as such in the descriptions that follow.

The following constant specifies the version number of the current version:

MQGMO_CURRENT_VERSION
Current version of get-message options structure.

This is always an input field. The initial value of this field is MQGMO_VERSION_1.

Options (MQLONG)
Options that control the action of MQGET.

Zero or more of the options described below can be specified. If more than one is required the values can be:

- Added together (do not add the same constant more than once), or
- Combined using the bitwise OR operation (if the programming language supports bit operations).

Combinations of options that are not valid are noted; all other combinations are valid. The following options are described:

MQGMO_WAIT
MQGMO_NO_WAIT
MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_NO_SYNCPOINT
MQGMO_MARK_SKIP_BACKOUT
MQGMO_BROWSE_FIRST
MQGMO_BROWSE_NEXT
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_MSG_UNDER_CURSOR
MQGMO_LOCK
MQGMO_UNLOCK
MQGMO_ACCEPT_TRUNCATED_MSG

MQGMO_SET_SIGNAL
MQGMO_FAIL_IF_QUIESCING
MQGMO_CONVERT
MQGMO_LOGICAL_ORDER
MQGMO_COMPLETE_MSG
MQGMO_ALL_MSGS_AVAILABLE
MQGMO_ALL_SEGMENTS_AVAILABLE
MQGMO_NONE

MQGMO_WAIT
Wait for message to arrive.

The application is to wait until a suitable message arrives. The maximum time the application waits is specified in *WaitInterval*.

If MQGET requests are inhibited, or MQGET requests become inhibited while waiting, the wait is canceled and the call completes with MQCC_FAILED and reason code MQRC_GET_INHIBITED, regardless of whether there are suitable messages on the queue.

This option can be used with the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT options.

On MVS/ESA, if it is desirable for the application to proceed with other work while waiting for the message to arrive, consider using the signal option (MQGMO_SET_SIGNAL) instead. However this option is environment specific, and so should not be used by applications which are intended to be portable between different environments.

If several applications are waiting on the same shared queue, the application, or applications, that are activated when a suitable message arrives are described below.

Note: In the description below, a *browse* MQGET call is one which specifies one of the browse options, but *not* MQGMO_LOCK; an MQGET call specifying the MQGMO_LOCK option is treated as a *nonbrowse* call.

- If one or more nonbrowse MQGET calls is waiting, one is activated.
- If one or more browse MQGET calls is waiting, but no nonbrowse MQGET calls are waiting, all are activated.
- If one or more nonbrowse MQGET calls, and one or more browse MQGET calls are waiting, one nonbrowse MQGET call is activated, and none, some, or all of the browse MQGET calls. (The number of browse MQGET calls activated cannot be predicted, because it depends on the scheduling considerations of the operating system, and other factors.)

If more than one nonbrowse MQGET call is waiting on the same shared queue, only one is activated; in this situation the queue manager attempts to give priority to waiting nonbrowse calls in the following order:

1. Specific get-wait requests that can be satisfied only by certain messages, for example, ones with a specific *MsgId* or *CorrelId* (or both).

2. General get-wait requests that can be satisfied by any message.

The following points should be noted:

- Within the first category, no additional priority is given to more specific get-wait requests, for example those that specify both *MsgId* and *CorrelId*.
- Within either category, it cannot be predicted which application is selected. In particular, the application waiting longest is not necessarily the one selected.
- Path length, and priority-scheduling considerations of the operating system, can mean that a waiting application of lower operating system 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.

On MVS/ESA, if there is more than one MQGET call waiting for the same message, with a mixture of wait and signal options, each waiting call is considered equally. It is an error to specify MQGMO_SET_SIGNAL with MQGMO_WAIT. It is also an error to specify this option with a queue handle for which a signal is outstanding.

MQGMO_WAIT is ignored if specified with MQGMO_BROWSE_MSG_UNDER_CURSOR or MQGMO_MSG_UNDER_CURSOR; no error is raised.

On Tandem NSK, only one nonbrowse MQGET call is activated for any given message. Nonbrowse MQGET calls are not given priority over browse MQGET calls waiting for the same message.

MQGMO_NO_WAIT
: Return immediately if no suitable message.

  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_SYNCPOINT
: Get message with syncpoint control.

  The request is to operate within the normal unit of work protocols. The message is marked as being unavailable to other applications, but it is deleted from the queue only when the unit of work is committed. The message is made available again if the unit of work is backed out.

  If neither this option nor MQGMO_NO_SYNCPOINT is specified, the inclusion of the get request in unit of work protocols is determined by the environment.

  - On MVS/ESA, the get request is within a unit of work.
  - In all other environments, the get request is not within a unit of work.

  Because of these differences, an application which is intended to be portable should not allow this option to default; either

MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT should be specified explicitly.

This option is not valid with any of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_BROWSE_NEXT
MQGMO_LOCK
MQGMO_NO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_UNLOCK

MQGMO_SYNCPOINT_IF_PERSISTENT
Get message with syncpoint control if message is persistent.

The request is to operate within the normal unit of work protocols, but *only* if the message retrieved is persistent. A persistent message has the value MQPER_PERSISTENT in the *Persistence* field in MQMD.

- If the message is persistent, the queue manager processes the call as though the application had specified MQGMO_SYNCPOINT (see above for details).
- If the message is not persistent, the queue manager processes the call as though the application had specified MQGMO_NO_SYNCPOINT (see below for details).

This option is not valid with any of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_BROWSE_NEXT
MQGMO_COMPLETE_MSG
MQGMO_MARK_SKIP_BACKOUT
MQGMO_NO_SYNCPOINT
MQGMO_SYNCPOINT
MQGMO_UNLOCK

This option is supported in the following environments: AIX, DOS client, HP-UX, MVS/ESA, OS/2, OS/400, Sun Solaris, Windows client, 32-bit Windows, Windows NT.

MQGMO_NO_SYNCPOINT
Get message without syncpoint control.

The request is to operate outside the normal unit of work protocols. The message is deleted from the queue immediately (unless this is a browse request). The message cannot be made available again by backing out a unit of work.

This option is assumed if MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT is specified.

If neither this option nor MQGMO_SYNCPOINT is specified, the inclusion of the get request in unit of work protocols is determined by the environment.

- On MVS/ESA, the get request is within a unit of work.

- In all other environments, the get request is not within a unit of work.

Because of these differences, an application which is intended to be portable should not allow this option to default; either MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT should be specified explicitly.

This option is not valid with any of the following options:

MQGMO_MARK_SKIP_BACKOUT
MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT

On Tandem NSK, if MQPUT is issued outside a Tandem TMF transaction *without* the MQPMO_NO_SYNCPOINT option, the reason code MQRC_UNIT_OF_WORK_NOT_STARTED is returned.

MQGMO_MARK_SKIP_BACKOUT
Mark the get request as skipping backout.

This option allows a unit of work to be backed out, but without reinstating on the queue the message that was marked with this option.

When an application requests the backout of a unit of work containing a get request, a message that was retrieved using this option is not restored to its previous state. (Other resource updates, however, are still backed out.) Instead, the message is treated as if it had been retrieved by a get request *without* this option, in a new unit of work started by the backout request.

This is useful if a message is retrieved by your application, but only after some resource updates have been made does it become apparent that the unit of work cannot complete successfully. A normal backout, if this option had not been specified, would cause the message to be reinstated on the queue, so that the same sequence of events would occur when the message was next retrieved. Using this option on the original MQGET, however, means that the backout will cause the updates to the other resources to be backed out, as is required, but the message is treated as if it had been retrieved under a new unit of work. The application can now perform some exception handling, such as informing the originator that the message has been discarded, and commit this new unit of work, which causes the message to be removed from the queue.

This option has an effect only if the unit of work containing the get request is terminated by an application request to back it out. (Such requests use calls or commands that depend on the environment.) This option has no effect if the unit of work containing the get request is backed out for any other reason (for example, the abend of a transaction or the system). In this situation, any message retrieved using this option is backed out on to the queue in the same way as messages retrieved without this option.

Notes:

1. If you have not applied IMS APAR PN60855 (or PN57124 for IMS V4), an IMS MPP or BMP application, returning a message obtained with the MQGMO_MARK_SKIP_BACKOUT option to the queue, must issue an MQ call (any MQ call will do) in between the two ROLB commands.
2. A CICS application, returning a message obtained with the MQGMO_MARK_SKIP_BACKOUT option to the queue, must issue an MQ call (any MQ call will do) in between the two EXEC CICS SYNCPOINT ROLLBACK commands.

Within a unit of work, there can be only one get request marked as skipping backout, as well as none or several unmarked get requests.

If this option is specified, MQGMO_SYNCPOINT must also be specified. MQGMO_MARK_SKIP_BACKOUT is not valid with any of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_BROWSE_NEXT
MQGMO_LOCK
MQGMO_NO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_UNLOCK

This option is not supported in the following environments: OpenVMS, OS/2, OS/400, Tandem NSK, UNIX systems, 16-bit Windows, 32-bit Windows, Windows NT.

MQGMO_BROWSE_FIRST
Browse from start of queue.

When a queue is opened with the MQOO_BROWSE option, a browse cursor is established, positioned logically before the first message on the queue. Subsequent MQGET calls specifying the MQGMO_BROWSE_FIRST, MQGMO_BROWSE_NEXT or MQGMO_BROWSE_MSG_UNDER_CURSOR option can be used to retrieve messages from the queue nondestructively. The browse cursor marks the position, within the messages on the queue, from which the next MQGET call with MQGMO_BROWSE_NEXT will search for a suitable message.

An MQGET call with MQGMO_BROWSE_FIRST causes the previous position of the browse cursor to be ignored. The first message on the queue that satisfies the conditions specified in the message descriptor is retrieved. The message remains on the queue, and the browse cursor is positioned on this message.

After this call, the browse cursor is positioned on the message that has been returned. If the message is removed from the queue before the next MQGET call with MQGMO_BROWSE_NEXT is issued, the browse cursor remains at the position in the queue that the message occupied, even though that position is now empty.

The MQGMO_MSG_UNDER_CURSOR option can subsequently be used with a nonbrowse MQGET call if required, to remove the message from the queue.

Note that the browse cursor is not moved by a nonbrowse MQGET call using the same *Hobj* handle. Nor is it moved by a browse MQGET call that returns a completion code of MQCC_FAILED, or a reason code of MQRC_TRUNCATED_MSG_FAILED.

The MQGMO_LOCK option can be specified together with this option, to cause the message that is browsed to be locked.

MQGMO_BROWSE_FIRST can be specified with any valid combination of the MQGMO_* and MQMO_* options that control the processing of messages in groups and segments of logical messages.

If MQGMO_LOGICAL_ORDER is specified, the messages are browsed in logical order. If that option is omitted, the messages are browsed in physical order. When MQGMO_BROWSE_FIRST is specified, it is possible to switch between logical order and physical order, but subsequent MQGET calls using MQGMO_BROWSE_NEXT must browse the queue in the same order as the most-recent call that specified MQGMO_BROWSE_FIRST for the queue handle.

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue is separate from the group and segment information that the queue manager retains for MQGET calls that remove messages from the queue. When MQGMO_BROWSE_FIRST is specified, the queue manager ignores the group and segment information for browsing, and scans the queue as though there were no current group and no current logical message. If the MQGET call is successful (completion code MQCC_OK or MQCC_WARNING), the group and segment information for browsing is set to that of the message returned; if the call fails, the group and segment information remains the same as it was prior to the call.

This option is not valid with any of the following options:

MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_BROWSE_NEXT
MQGMO_MARK_SKIP_BACKOUT
MQGMO_MSG_UNDER_CURSOR
MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_UNLOCK

It is also an error if the queue was not opened for browse.

MQGMO_BROWSE_NEXT
Browse from current position in queue.

The browse cursor is advanced to the next message on the queue that satisfies the selection criteria specified on the MQGET call. The message is returned to the application, but remains on the queue.

After a queue has been opened for browse, the first browse call using the handle has the same effect whether it specifies the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT option.

If the message is removed from the queue before the next MQGET call with MQGMO_BROWSE_NEXT is issued, the browse cursor

logically remains at the position in the queue that the message occupied, even though that position is now empty.

Messages are stored on the queue in one of two ways:

- FIFO within priority (MQMDS_PRIORITY), or
- FIFO *regardless* of priority (MQMDS_FIFO)

The *MsgDeliverySequence* queue attribute indicates which method applies (see “Attributes for local queues and model queues” on page 348 for details).

If the queue has a *MsgDeliverySequence* of MQMDS_PRIORITY, and a message arrives on the queue that is of a higher priority than the one currently pointed to by the browse cursor, that message will not be found during the current sweep of the queue using MQGMO_BROWSE_NEXT. It can only be found after the browse cursor has been reset with MQGMO_BROWSE_FIRST (or by reopening the queue).

The MQGMO_MSG_UNDER_CURSOR option can subsequently be used with a nonbrowse MQGET call if required, to remove the message from the queue.

Note that the browse cursor is not moved by nonbrowse MQGET calls using the same *Hobj* handle.

The MQGMO_LOCK option can be specified together with this option, to cause the message that is browsed to be locked.

MQGMO_BROWSE_NEXT can be specified with any valid combination of the MQGMO_★ and MQMO_★ options that control the processing of messages in groups and segments of logical messages.

If MQGMO_LOGICAL_ORDER is specified, the messages are browsed in logical order. If that option is omitted, the messages are browsed in physical order. When MQGMO_BROWSE_FIRST is specified, it is possible to switch between logical order and physical order, but subsequent MQGET calls using MQGMO_BROWSE_NEXT must browse the queue in the same order as the most-recent call that specified MQGMO_BROWSE_FIRST for the queue handle. The call fails with reason code MQRC_INCONSISTENT_BROWSE if this condition is not satisfied.

Note: Special care is needed if an MQGET call is used to browse *beyond the end* of a message group (or logical message not in a group) when MQGMO_LOGICAL_ORDER is not specified. For example, if the last message in the group happens to *precede* the first message in the group on the queue, using MQGMO_BROWSE_NEXT to browse beyond the end of the group, specifying MQMO_MATCH_MSG_SEQ_NUMBER with *MsgSeqNumber* set to 1 (to find the first message of the next group) would return again the first message in the group already browsed. This could happen immediately, or a number of MQGET calls later (if there are intervening groups).

The possibility of an infinite loop can be avoided by opening the queue *twice* for browse:

- Use the first handle to browse only the first message in each group.
- Use the second handle to browse only the messages within a specific group.
- Use the MQMO_★ options to move the second browse cursor to the position of the first browse cursor, before browsing the messages in the group.
- Do not use MQGMO_BROWSE_NEXT to browse beyond the end of a group.

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue is separate from the group and segment information that it retains for MQGET calls that remove messages from the queue.

This option is not valid with any of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_MARK_SKIP_BACKOUT
MQGMO_MSG_UNDER_CURSOR
MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_UNLOCK

It is also an error if the queue was not opened for browse.

MQGMO_BROWSE_MSG_UNDER_CURSOR
Browse message under browse cursor.

This option causes the message pointed to by the browse cursor to be retrieved nondestructively, regardless of the MQMO_★ options specified in the *MatchOptions* field in MQGMO.

On MVS/ESA, this option is not supported.

The message pointed to by the browse cursor is the one that was last retrieved using either the MQGMO_BROWSE_FIRST or the MQGMO_BROWSE_NEXT option. The call fails if neither of these calls has been issued for this queue since it was opened, or if the message that was under the browse cursor has since been retrieved destructively.

The position of the browse cursor is not changed by this call.

The MQGMO_MSG_UNDER_CURSOR option can subsequently be used with a nonbrowse MQGET call if required, to remove the message from the queue.

Note that the browse cursor is not moved by a nonbrowse MQGET call using the same *Hobj* handle. Nor is it moved by a browse MQGET call that returns a completion code of MQCC_FAILED, or a reason code of MQRC_TRUNCATED_MSG_FAILED.

If MQGMO_BROWSE_MSG_UNDER_CURSOR is specified *with* MQGMO_LOCK:

- If there is already a message locked, it must be the one under the cursor, so that is returned *without* unlocking and relocking it; the message remains locked.
- If there is no locked message, the message under the browse cursor (if there is one) is locked and returned to the application; if there is no message under the browse cursor the call fails.

If MQGMO_BROWSE_MSG_UNDER_CURSOR is specified *without* MQGMO_LOCK:

- If there is already a message locked, it must be the one under the cursor. This message is returned to the application *and then unlocked*. Because the message is now unlocked, there is no guarantee that it can be browsed again, or retrieved destructively (it may be retrieved destructively by another application getting messages from the queue).
- If there is no locked message, the message under the browse cursor (if there is one) is returned to the application; if there is no message under the browse cursor the call fails.

If MQGMO_COMPLETE_MSG is specified with MQGMO_BROWSE_MSG_UNDER_CURSOR, the browse cursor must identify a message whose *Offset* field in MQMD is zero. If this condition is not satisfied, the call fails with reason code MQRC_INVALID_MSG_UNDER_CURSOR.

The group and segment information that the queue manager retains for MQGET calls that browse messages on the queue is separate from the group and segment information that it retains for MQGET calls that remove messages from the queue.

This option is not valid with any of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_NEXT
MQGMO_MARK_SKIP_BACKOUT
MQGMO_MSG_UNDER_CURSOR
MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_UNLOCK

It is also an error if the queue was not opened for browse.

MQGMO_MSG_UNDER_CURSOR
Get message under browse cursor.

This option causes the message pointed to by the browse cursor to be retrieved, regardless of the MQMO_★ options specified in the *MatchOptions* field in MQGMO. The message is removed from the queue.

The message pointed to by the browse cursor is the one that was last retrieved using either the MQGMO_BROWSE_FIRST or the MQGMO_BROWSE_NEXT option.

If MQGMO_COMPLETE_MSG is specified with MQGMO_MSG_UNDER_CURSOR, the browse cursor must identify a message whose *Offset* field in MQMD is zero. If this condition is

not satisfied, the call fails with reason code MQRC_INVALID_MSG_UNDER_CURSOR.

This option is not valid with any of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_BROWSE_NEXT
MQGMO_UNLOCK

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.

MQGMO_LOCK
Lock message.

This option locks the message that is browsed, so that the message becomes invisible to any other handle open for the queue. The option can be specified only if one of the following options is also specified:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_NEXT
MQGMO_BROWSE_MSG_UNDER_CURSOR

Only one message can be locked per handle, but this can be a logical message or a physical message:

- If MQGMO_COMPLETE_MSG is specified, all of the message segments that comprise the logical message are locked to the queue handle (provided that they are all present on the queue and available for retrieval).
- If MQGMO_COMPLETE_MSG is *not* specified, only a single physical message is locked to the queue handle. If this message happens to be a segment of a logical message, the locked segment prevents other applications using MQGMO_COMPLETE_MSG to retrieve or browse the logical message.

The locked message is always the one under the browse cursor, and the message can be removed from the queue by a later MQGET call that specifies the MQGMO_MSG_UNDER_CURSOR option. Other MQGET calls for that queue handle can also remove the message (for example, a call that specifies the message identifier of the locked message).

If MQCC_FAILED is returned (or MQCC_WARNING with MQRC_TRUNCATED_MSG_FAILED), no message is locked.

If the application decides not to remove the message from the queue, the lock is released by:

- Issuing another MQGET call for this handle, with either MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT specified (with or without MQGMO_LOCK); the message is unlocked if the call completes with MQCC_OK or MQCC_WARNING, but remains locked if the call completes with MQCC_FAILED. However, the following exceptions apply:

- The message *is not* unlocked if MQCC_WARNING is returned with MQRC_TRUNCATED_MSG_FAILED.
- The message *is* unlocked if MQCC_FAILED is returned with MQRC_NO_MSG_AVAILABLE.

If MQGMO_LOCK is also specified, the new message is locked. If MQGMO_LOCK is not specified, there is no locked message after the call.

If MQGMO_WAIT is specified, and no message is immediately available, the unlock on the original message occurs before the start of the wait (providing the call is otherwise free from error).

- Issuing another MQGET call for this handle, with MQGMO_BROWSE_MSG_UNDER_CURSOR (without MQGMO_LOCK); the message is unlocked if the call completes with MQCC_OK or MQCC_WARNING, but remains locked if the call completes with MQCC_FAILED. However, the following exception applies:
  - The message *is not* unlocked if MQCC_WARNING is returned with MQRC_TRUNCATED_MSG_FAILED.
- Issuing another MQGET call for this handle with MQGMO_UNLOCK.
- Issuing an MQCLOSE call for this handle (either explicitly, or implicitly by the application ending).

No special open option is required to specify this option, other than MQOO_BROWSE, which is needed in order to specify the accompanying browse option.

This option is not valid with any of the following options:

MQGMO_MARK_SKIP_BACKOUT
MQGMO_SYNCPOINT
MQGMO_SYNCPOINT_IF_PERSISTENT
MQGMO_UNLOCK

This option is not supported in the following environments: MVS/ESA, Tandem NSK, 16-bit Windows, 32-bit Windows.

MQGMO_UNLOCK
Unlock message.

The message to be unlocked must have been previously locked by an MQGET call with the MQGMO_LOCK option. If there is no message locked for this handle, the call completes with MQCC_WARNING and MQRC_NO_MSG_LOCKED.

The *MsgDesc*, *BufferLength*, *Buffer*, and *DataLength* parameters are not checked or altered if MQGMO_UNLOCK is specified. No message is returned in *Buffer*.

No special open option is required to specify this option (although MQOO_BROWSE is needed to issue the lock request in the first place).

This option is not valid with any options *except* the following:

MQGMO_NO_WAIT

MQGMO_NO_SYNCPOINT

Both of these options are assumed whether specified or not.

This option is not supported in the following environments: MVS/ESA, Tandem NSK, 16-bit Windows, 32-bit Windows.

MQGMO_ACCEPT_TRUNCATED_MSG
Allow truncation of message data.

If the message buffer is too small to hold the complete message, this option allows the MQGET call to fill the buffer with as much of the message as the buffer can hold, issue a warning completion code, and complete its processing. This means:

- When browsing messages, the browse cursor is advanced to the returned message.
- When removing messages, the returned message is removed from the queue.
- Reason code MQRC_TRUNCATED_MSG_ACCEPTED is returned if no other error occurs.

Without this option, the buffer is still filled with as much of the message as it can hold, a warning completion code is issued, but processing is not completed. This means:

- When browsing messages, the browse cursor is not advanced.
- When removing messages, the message is not removed from the queue.
- Reason code MQRC_TRUNCATED_MSG_FAILED is returned if no other error occurs.

MQGMO_SET_SIGNAL
Request signal to be set.

This option is used in conjunction with the *Signal1* and *Signal2* fields to allow applications to proceed with other work while waiting for a message to arrive, and also (if suitable operating system facilities are available) to wait for messages arriving on more than one queue.

The MQGMO_SET_SIGNAL option is environment specific, and should not be used by applications which are intended to be portable.

If a currently available message satisfies the criteria specified in the message descriptor, or if a parameter error or other synchronous error is detected, the call completes in the same way as if this option had not been specified.

If no message satisfying the criteria specified in the message descriptor is currently available, control returns to the application without waiting for a message to arrive. The output fields in the message descriptor and the output parameters of the MQGET call are not set, other than the *CompCode* and *Reason* parameters (which are set to MQCC_WARNING and MQRC_SIGNAL_REQUEST_ACCEPTED respectively). When a

suitable message arrives subsequently, the signal is delivered in a manner dependent on the environment:

- On MVS/ESA, the signal is delivered by posting the ECB.
- On 32-bit Windows, a Windows message is sent to the application.

The caller should then reissue the MQGET call to retrieve the message. The application can wait for this signal, using functions provided by the operating system.

If the operating system provides a multiple wait mechanism, the application can use this technique to wait for a message arriving on any one of several queues.

If a nonzero *WaitInterval* is specified, after this time the signal is delivered. The wait may also be canceled by the queue manager, in which case again the signal is delivered.

If more than one MQGET call has set a signal for the same message, the order in which applications are activated is the same as that described for MQGMO_WAIT.

If there is more than one MQGET call waiting for the same message, with a mixture of wait and signal options, each waiting call is considered equally.

Under certain conditions it is possible for a message to be retrieved by the MQGET call, *and* for a signal resulting from the arrival of the same message to be delivered. When a signal is delivered, an application must be prepared for no message to be available.

A given handle can have no more than one signal outstanding.

This option is supported *only* in the following environments:
| MVS/ESA, Tandem NSK, 32-bit Windows.

On MVS/ESA and 32-bit Windows, MQGMO_SET_SIGNAL is not valid with any of the following options:

MQGMO_UNLOCK
MQGMO_WAIT

| On Tandem NSK, MQGMO_SET_SIGNAL is not valid with any of the
| following options:

| MQGMO_BROWSE_FIRST
| MQGMO_BROWSE_NEXT
| MQGMO_BROWSE_MSG_UNDER_CURSOR
| MQGMO_LOCK
| MQGMO_UNLOCK
| MQGMO_WAIT

MQGMO_FAIL_IF_QUIESCING
Fail if queue manager is quiescing.

This option forces the MQGET call to fail if the queue manager is in the quiescing state.

On MVS/ESA, this option also forces the MQGET call to fail if the connection (for a CICS or IMS application) is in the quiescing state.

If this option is specified together with MQGMO_WAIT or MQGMO_SET_SIGNAL, and the wait or signal is outstanding at the time the queue manager enters the quiescing state:

- The wait is canceled and the call returns completion code MQCC_FAILED with reason code MQRC_Q_MGR_QUIESCING or MQRC_CONNECTION_QUIESCING.
- The signal is canceled with an environment-specific signal completion code.

  On MVS/ESA, the signal completes with event completion code MQEC_Q_MGR_QUIESCING or MQEC_CONNECTION_QUIESCING.

If MQGMO_FAIL_IF_QUIESCING is not specified and the queue manager or connection enters the quiescing state, the wait or signal is not canceled.

In the following environments, this option is accepted but ignored: 16-bit Windows, 32-bit Windows.

MQGMO_CONVERT
Convert message data.

This option requests that the application data in the message should be converted, to conform to the *CodedCharSetId* and *Encoding* values specified in the *MsgDesc* parameter on the MQGET call, before the data is copied to the *Buffer* parameter.

The *Format* field specified when the message was put is assumed by the conversion process to identify the nature of the data in the message. Conversion of the message data is by the queue manager for built-in formats, and by a user-written exit for other formats. See Appendix D, “Data-conversion” on page 495 for details of the data-conversion exit.

- If conversion is performed successfully, the *CodedCharSetId* and *Encoding* fields specified in the *MsgDesc* parameter are unchanged on return from the MQGET call.
- If conversion cannot be performed successfully (but the MQGET call otherwise completes without error), the message data is returned unconverted, and the *CodedCharSetId* and *Encoding* fields in *MsgDesc* are set to the values for the unconverted message. The completion code is MQCC_WARNING in this case.

In either case, therefore, these fields describe the character-set identifier and encoding of the message data that is returned in the *Buffer* parameter.

See the *Format* field described in “MQMD – Message descriptor” on page 98 for a list of format names for which the queue manager performs the conversion.

This option is not supported in the following environments: MVS/ESA using CICS version 2, 16-bit Windows, 32-bit Windows.

Group and segment options: The options described below control the way that messages in groups and segments of logical messages are

returned by the MQGET call. The following definitions may be of help in understanding these options:

Physical message
This is the smallest unit of information that can be placed on or removed from a queue; it often corresponds to the information specified or retrieved on a single MQPUT, MQPUT1, or MQGET call. Every physical message has its own message descriptor (MQMD). Generally, physical messages are distinguished by differing values for the message identifier (*MsgId* field in MQMD), although this is not enforced by the queue manager.

Logical message
This is a single unit of application information. In the absence of system constraints, a logical message would be the same as a physical message. But where logical messages are extremely large, system constraints may make it advisable or necessary to split a logical message into two or more physical messages, called *segments*.

A logical message that has been segmented consists of two or more physical messages that have the same nonnull group identifier (*GroupId* field in MQMD), and the same message sequence number (*MsgSeqNumber* field in MQMD). The segments are distinguished by differing values for the segment offset (*Offset* field in MQMD), which gives the offset of the data in the physical message from the start of the data in the logical message. Because each segment is a physical message, the segments in a logical message usually have differing message identifiers.

A logical message that has not been segmented, but for which segmentation has been permitted by the sending application, also has a nonnull group identifier, although in this case there is only one physical message with that group identifier if the logical message does not belong to a message group. Logical messages for which segmentation has been inhibited by the sending application have a null group identifier (MQGI_NONE), unless the logical message belongs to a message group.

Message group
This is a set of one or more logical messages that have the same nonnull group identifier. The logical messages in the group are distinguished by differing values for the message sequence number, which is an integer in the range 1 through n, where n is the number of logical messages in the group. If one or more of the logical messages is segmented, there will be more than n physical messages in the group.

MQGMO_LOGICAL_ORDER
Messages in groups and segments of logical messages are returned in logical order.

This option controls the order in which messages are returned by *successive* MQGET calls for the queue handle. The option must be specified on each of those calls in order to have an effect.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

If MQGMO_LOGICAL_ORDER is specified for successive MQGET calls for the queue handle, messages in groups are returned in the order given by their message sequence numbers, and segments of logical messages are returned in the order given by their segment offsets. This order may be different from the order in which those messages and segments occur on the queue.

Note: Specifying MQGMO_LOGICAL_ORDER has no adverse consequences on messages that do not belong to groups and that are not segments. In effect, such messages are treated as though each belonged to a message group consisting of only one message. Thus it is perfectly safe to specify MQGMO_LOGICAL_ORDER when retrieving messages from queues that may contain a mixture of messages in groups, message segments, and unsegmented messages not in groups.

To return the messages in the required order, the queue manager retains the group and segment information between successive MQGET calls. This information identifies the current message group and current logical message for the queue handle, the current position within the group and logical message, and whether the messages are being retrieved within a unit of work. Because the queue manager retains this information, the application does not need to set the group and segment information prior to each MQGET call. Specifically, it means that the application does not need to set the *GroupId*, *MsgSeqNumber*, and *Offset* fields in MQMD. However, the application does need to set the MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT option correctly on each call.

When the queue is opened, there is no current message group and no current logical message. A message group becomes the current message group when a message that has the MQMF_MSG_IN_GROUP flag is returned by the MQGET call. With MQGMO_LOGICAL_ORDER specified on successive calls, that group remains the current group until a message is returned that has:

- MQMF_LAST_MSG_IN_GROUP without MQMF_SEGMENT (that is, the last logical message in the group is not segmented), or
- MQMF_LAST_MSG_IN_GROUP with MQMF_LAST_SEGMENT (that is, the message returned is the last segment of the last logical message in the group).

When such a message is returned, the message group is terminated, and on successful completion of that MQGET call there is no longer a current group. In a similar way, a logical message becomes the current logical message when a message that has the MQMF_SEGMENT flag is returned by the MQGET call, and that logical message is terminated when the message that has the MQMF_LAST_SEGMENT flag is returned.

If no selection criteria are specified, successive MQGET calls return (in the correct order) the messages for the first message group on the queue, then the messages for the second message group, and so on, until there are no more messages available. It is possible to select the particular message groups returned by specifying one or more of the following options in the *MatchOptions* field:

MQMO_MATCH_MSG_ID
MQMO_MATCH_CORREL_ID
MQMO_MATCH_GROUP_ID

However, these options are effective only when there is no current message group or logical message; see the *MatchOptions* field described in “MQGMO – Get-message options” on page 56 for further details.

Table 29 on page 75 shows the values of the *MsgId*, *CorrelId*, *GroupId*, *MsgSeqNumber*, and *Offset* fields that the queue manager looks for when attempting to find a message to return on the MQGET call. This applies both to removing messages from the queue, and browsing messages on the queue. The abbreviated column headings have the following meanings:

- **LOG ORD** means the MQGMO_LOGICAL_ORDER option.
- **Cur grp** means that a current message group exists prior to the call.
- **Cur log msg** means that a current logical message exists prior to the call.

In the table:

- “(√)” indicates that the row applies whether or not there is a √ in that column.
- “Previous” denotes the value returned for that field in the previous message for the queue handle.

Table 29. MQGET options relating to messages in groups and segments of logical messages

Options you specify	Group and log-msg status prior to call		Values the queue manager looks for				
LOG ORD	**Cur grp**	**Cur log msg**	*MsgId*	*CorrelId*	*GroupId*	*MsgSeqNumber*	*Offset*
√			Controlled by *MatchOptions*	Controlled by *MatchOptions*	Controlled by *MatchOptions*	1	0
√		√	Any message identifier	Any correlation identifier	Previous group identifier	1	Previous offset + previous segment length
√	√		Any message identifier	Any correlation identifier	Previous group identifier	Previous sequence number + 1	0
√	√	√	Any message identifier	Any correlation identifier	Previous group identifier	Previous sequence number	Previous offset + previous segment length
	(√)	(√)	Controlled by *MatchOptions*	Controlled by *MatchOptions*	Controlled by *MatchOptions*	Controlled by *MatchOptions*	Controlled by *MatchOptions*

When multiple message groups are present on the queue and eligible for return, the groups are returned in the order determined by the position on the queue of the first segment of the first logical message in each group (that is, the physical messages that have message sequence numbers of 1, and offsets of 0, determine the order in which eligible groups are returned).

The MQGMO_LOGICAL_ORDER option affects units of work as follows:

- If the first logical message or segment in a group is retrieved within a unit of work, all of the other logical messages and segments in the group must be retrieved within a unit of work, if the same queue handle is used. However, they need not be retrieved within the same unit of work. This allows a message group consisting of many physical messages to be split across two or more consecutive units of work for the queue handle.
- If the first logical message or segment in a group is *not* retrieved within a unit of work, none of the other logical messages and segments in the group can be retrieved within a unit of work, if the same queue handle is used.

If these conditions are not satisfied, the MQGET call fails with reason code MQRC_INCONSISTENT_UOW.

When MQGMO_LOGICAL_ORDER is specified, the MQGMO supplied on the MQGET call must not be less than MQGMO_VERSION_2, and the MQMD must not be less than MQMD_VERSION_2. If this condition is not satisfied, the call fails with reason code MQRC_WRONG_GMO_VERSION or MQRC_WRONG_MD_VERSION, as appropriate.

If MQGMO_LOGICAL_ORDER is *not* specified for successive MQGET calls for the queue handle, messages are returned without regard for whether they belong to message groups, or whether they are segments of logical messages. This means that messages or segments from a particular group or logical message may be returned out of order, or they may be intermingled with messages or segments from other groups or logical messages, or with messages that are not in groups and are not segments. In this situation, the particular messages that are returned by successive MQGET calls is controlled by the MQMO_* options specified on those calls (see the *MatchOptions* field described in "MQGMO – Get-message options" on page 56 for details of these options).

This is the technique that can be used to restart a message group or logical message in the middle, after a system failure has occurred. When the system restarts, the application can set the *GroupId*, *MsgSeqNumber*, *Offset*, and *MatchOptions* fields to the appropriate values, and then issue the MQGET call with MQGMO_SYNCPOINT or MQGMO_NO_SYNCPOINT set as desired, but *without* specifying MQGMO_LOGICAL_ORDER. If this call is successful, the queue manager retains the group and segment information, and subsequent MQGET calls using that queue handle can specify MQGMO_LOGICAL_ORDER as normal.

The group and segment information that the queue manager retains for the MQGET call is separate from the group and segment information that it retains for the MQPUT call. In addition, the queue manager retains separate information for:

- MQGET calls that remove messages from the queue.
- MQGET calls that browse messages on the queue.

For any given queue handle, the application is free to mix MQGET calls that specify MQGMO_LOGICAL_ORDER with MQGET calls that do not, but the following points should be noted:

- Each successful MQGET call that does *not* specify MQGMO_LOGICAL_ORDER causes the queue manager to set the saved group and segment information to the values corresponding to the message returned; this replaces the existing group and segment information retained by the queue manager for the queue handle. Only the information appropriate to the action of the call (browse or remove) is modified.
- If MQGMO_LOGICAL_ORDER is *not* specified, the call does not fail if there is a current message group or logical message, but the message or segment retrieved is not the next one in the group or logical message. The call may however succeed with an MQCC_WARNING completion code. Table 30 on page 77 shows the various cases that can arise. In these cases, if the completion code is not MQCC_OK, the reason code is one of the following (as appropriate):

  MQRC_INCOMPLETE_GROUP
  MQRC_INCOMPLETE_MSG
  MQRC_INCONSISTENT_UOW

Note: The queue manager does not check the group and segment information when browsing a queue, or when closing a queue that was opened for browse but not input; in those cases the completion code is always MQCC_OK (assuming no other errors).

Table 30. Outcome when MQGET or MQCLOSE call not consistent with group and segment information

Current call	**Previous call**	
	MQGET with MQGMO_LOGICAL_ORDER	MQGET without MQGMO_LOGICAL_ORDER
MQGET with MQGMO_LOGICAL_ORDER	MQCC_FAILED	MQCC_FAILED
MQGET without MQGMO_LOGICAL_ORDER	MQCC_WARNING	MQCC_OK
MQCLOSE with an unterminated group or logical message	MQCC_WARNING	MQCC_OK

Applications that simply want to retrieve messages and segments in logical order are recommended to specify MQGMO_LOGICAL_ORDER, as this is the simplest option to use. This option relieves the application of the need to manage the group and segment information, because the queue manager manages that information. However, specialized applications may need more control than provided by the MQGMO_LOGICAL_ORDER option, and this can be achieved by not specifying that option. If this is done, the application must ensure that the *MsgId*, *CorrelId*, *GroupId*, *MsgSeqNumber*, and *Offset* fields in MQMD, and the MQMO_★ options in *MatchOptions* in MQGMO, are set correctly, prior to each MQGET call.

For example, an application that wants to *forward* physical messages that it receives, without regard for whether those messages are in groups or segments of logical messages, should *not* specify MQGMO_LOGICAL_ORDER. This is because in a complex network with multiple paths between sending and receiving queue managers, the physical messages may arrive out of order. By specifying neither MQGMO_LOGICAL_ORDER, nor the corresponding MQPMO_LOGICAL_ORDER on the MQPUT call, the forwarding application can retrieve and forward each physical message as soon as it arrives, without having to wait for the next one in logical order to arrive.

MQGMO_LOGICAL_ORDER can be specified with any of the other MQGMO_★ options, and with various of the MQMO_★ options in appropriate circumstances (see above).

MQGMO_COMPLETE_MSG
Only complete logical messages are retrievable.

This option specifies that only a complete logical message can be returned by the MQGET call. If the logical message is segmented, the queue manager reassembles the segments and returns the complete logical message to the application; the fact that the logical

message was segmented is not apparent to the application retrieving it.

Note: This is the only option that causes the queue manager to reassemble message segments. If not specified, segments are returned individually to the application if they are present on the queue (and they satisfy the other selection criteria specified on the MQGET call). Applications that do not wish to receive individual segments should therefore always specify MQGMO_COMPLETE_MSG.

To use this option, the application must provide a buffer which is big enough to accommodate the complete message, or specify the MQGMO_ACCEPT_TRUNCATED_MSG option.

If the queue contains segmented messages with some of the segments missing (perhaps because they have been delayed in the network and have not yet arrived), specifying MQGMO_COMPLETE_MSG prevents the retrieval of segments belonging to incomplete logical messages. However, those message segments still contribute to the value of the *CurrentQDepth* queue attribute; this means that there may be no retrievable logical messages, even though *CurrentQDepth* is greater than zero.

For *persistent* messages, the queue manager can reassemble the segments only within a unit of work:

- If the MQGET call is operating within a user-defined unit of work, that unit of work is used. If the call fails partway through the reassembly process, the queue manager reinstates on the queue any segments that were removed during reassembly. However, the failure does not prevent the unit of work being committed successfully.
- If the call is operating outside a user-defined unit of work, and there is no user-defined unit of work in existence, the queue manager creates a unit of work just for the duration of the call. If the call is successful, the queue manager commits the unit of work automatically (the application does not need to do this). If the call fails, the queue manager backs out the unit of work.
- If the call is operating outside a user-defined unit of work, but a user-defined unit of work *does* exist, the queue manager is unable to perform reassembly. If the message does not require reassembly, the call can still succeed. But if the message *does* require reassembly, the call fails with reason code MQRC_UOW_NOT_AVAILABLE.

For *nonpersistent* messages, the queue manager does not require a unit of work to be available in order to perform reassembly.

Each physical message which is a segment has its own message descriptor. For the segments constituting a single logical message, most of the fields in the message descriptor will be the same for all segments in the logical message – usually it is only the *MsgId*, *Offset*, and *MsgFlags* fields that differ between segments in the logical message. However, when segments take different paths through the network, and some of those paths have MCA sender conversion enabled, it is possible for the *CodedCharSetId* and

Encoding fields to differ between segments when the segments eventually arrive at the destination queue. A logical message consisting of segments in which the *CodedCharSetId* and/or *Encoding* fields differ cannot be reassembled by the queue manager into a single logical message. Instead, the queue manager reassembles and returns the first few consecutive segments at the start of the logical message that have the same character-set identifiers and encodings, and the MQGET call completes with completion code MQCC_WARNING and reason code MQRC_INCONSISTENT_CCSIDS or MQRC_INCONSISTENT_ENCODINGS, as appropriate. This happens regardless of whether MQGMO_CONVERT is specified. To retrieve the remaining segments, the application must reissue the MQGET call without the MQGMO_COMPLETE_MSG option, retrieving the segments one by one. MQGMO_LOGICAL_ORDER can be used to retrieve the remaining segments in order.

It is also possible for an application which puts segments to set other fields in the message descriptor to values that differ between segments. However, there is no advantage in doing this if the receiving application uses MQGMO_COMPLETE_MSG to retrieve the logical message. When the queue manager reassembles a logical message, it returns in the message descriptor the values from the message descriptor for the *first* segment; the only exception is the *MsgFlags* field, which the queue manager sets to indicate that the reassembled message is the only segment.

If MQGMO_COMPLETE_MSG is specified for a report message, the queue manager performs special processing. The queue manager checks the queue to see if all of the report messages of that report type relating to the different segments in the logical message are present on the queue. If they are, they can be retrieved as a single message by specifying MQGMO_COMPLETE_MSG. For this to be possible, either the report messages must be generated by a queue manager or MCA which supports segmentation, or the originating application must request at least 100 bytes of message data (that is, the appropriate MQRO_*_WITH_DATA or MQRO_*_WITH_FULL_DATA options must be specified). If less than the full amount of application data is present for a segment, the missing bytes are replaced by nulls in the report message returned.

If MQGMO_COMPLETE_MSG is specified with MQGMO_MSG_UNDER_CURSOR or MQGMO_BROWSE_MSG_UNDER_CURSOR, the browse cursor must be positioned on a message whose *Offset* field in MQMD has a value of 0. If this condition is not satisfied, the call fails with reason code MQRC_INVALID_MSG_UNDER_CURSOR.

MQGMO_COMPLETE_MSG implies MQGMO_ALL_SEGMENTS_AVAILABLE, which need not therefore be specified.

MQGMO_COMPLETE_MSG can be specified with any of the other MQGMO_* options apart from MQGMO_SYNCPOINT_IF_PERSISTENT, and with any of the MQMO_* options apart from MQMO_MATCH_OFFSET.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQGMO_ALL_MSGS_AVAILABLE
All messages in group must be available.

This option specifies that messages in a group become available for retrieval only when *all* messages in the group are available. If the queue contains message groups with some of the messages missing (perhaps because they have been delayed in the network and have not yet arrived), specifying MQGMO_ALL_MSGS_AVAILABLE prevents retrieval of messages belonging to incomplete groups. However, those messages still contribute to the value of the *CurrentQDepth* queue attribute; this means that there may be no retrievable message groups, even though *CurrentQDepth* is greater than zero. If there are no other messages that are retrievable, reason code MQRC_NO_MSG_AVAILABLE is returned after the specified wait interval (if any) has expired.

The processing of MQGMO_ALL_MSGS_AVAILABLE depends on whether MQGMO_LOGICAL_ORDER is also specified:

- If both options are specified, MQGMO_ALL_MSGS_AVAILABLE has an effect *only* when there is no current group or logical message. If there *is* a current group or logical message, MQGMO_ALL_MSGS_AVAILABLE is ignored. This means that MQGMO_ALL_MSGS_AVAILABLE can remain on when processing messages in logical order.
- If MQGMO_ALL_MSGS_AVAILABLE is specified without MQGMO_LOGICAL_ORDER, MQGMO_ALL_MSGS_AVAILABLE *always* has an effect. This means that the option must be turned off after the first message in the group has been removed from the queue, in order to be able to remove the remaining messages in the group.

If this option is not specified, messages belonging to groups can be retrieved even when the group is incomplete.

MQGMO_ALL_MSGS_AVAILABLE implies MQGMO_ALL_SEGMENTS_AVAILABLE, which need not therefore be specified.

MQGMO_ALL_MSGS_AVAILABLE can be specified with any of the other MQGMO_★ options, and with any of the MQMO_★ options.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQGMO_ALL_SEGMENTS_AVAILABLE
All segments in a logical message must be available.

This option specifies that segments in a logical message become available for retrieval only when *all* segments in the logical message are available. If the queue contains segmented messages with some of the segments missing (perhaps because they have been delayed in the network and have not yet arrived), specifying MQGMO_ALL_SEGMENTS_AVAILABLE prevents retrieval of

segments belonging to incomplete logical messages. However those segments still contribute to the value of the *CurrentQDepth* queue attribute; this means that there may be no retrievable logical messages, even though *CurrentQDepth* is greater than zero. If there are no other messages that are retrievable, reason code MQRC_NO_MSG_AVAILABLE is returned after the specified wait interval (if any) has expired.

The processing of MQGMO_ALL_SEGMENTS_AVAILABLE depends on whether MQGMO_LOGICAL_ORDER is also specified:

- If both options are specified, MQGMO_ALL_SEGMENTS_AVAILABLE has an effect *only* when there is no current logical message. If there *is* a current logical message, MQGMO_ALL_SEGMENTS_AVAILABLE is ignored. This means that MQGMO_ALL_SEGMENTS_AVAILABLE can remain on when processing messages in logical order.
- If MQGMO_ALL_SEGMENTS_AVAILABLE is specified without MQGMO_LOGICAL_ORDER, MQGMO_ALL_SEGMENTS_AVAILABLE *always* has an effect. This means that the option must be turned off after the first segment in the logical message has been removed from the queue, in order to be able to remove the remaining segments in the logical message.

If this option is not specified, message segments can be retrieved even when the logical message is incomplete.

While both MQGMO_COMPLETE_MSG and MQGMO_ALL_SEGMENTS_AVAILABLE require all segments to be available before any of them can be retrieved, the former returns the complete message, whereas the latter allows the segments to be retrieved one by one.

If MQGMO_ALL_SEGMENTS_AVAILABLE is specified for a report message, the queue manager performs special processing. The queue manager checks the queue to see if there is at least one report message for each of the segments that comprise the complete logical message. If there is, the MQGMO_ALL_SEGMENTS_AVAILABLE condition is satisfied. However, the queue manager does not check the *type* of the report messages present, and so there may be a mixture of report types in the report messages relating to the segments of the logical message. As a result, the success of MQGMO_ALL_SEGMENTS_AVAILABLE does not imply that MQGMO_COMPLETE_MSG will succeed. If there *is* a mixture of report types present for the segments of a particular logical message, those report messages must be retrieved one by one.

MQGMO_ALL_SEGMENTS_AVAILABLE can be specified with any of the other MQGMO_★ options, and with any of the MQMO_★ options.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQGMO_NONE
No options specified.

This value can be used to indicate that no other options have been specified; all options assume their default values. MQGMO_NONE is defined to aid program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

The initial value of the *Options* field is MQGMO_NO_WAIT.

WaitInterval (MQLONG)
Wait interval.

This is the approximate time, expressed in milliseconds, that the MQGET call waits for a suitable message to arrive (that is, a message satisfying the selection criteria specified in the *MsgDesc* parameter of the MQGET call; see the *MsgId* field described in "MQMD – Message descriptor" on page 98 for more details). If no suitable message has arrived after this time has elapsed, the call completes with MQCC_FAILED and reason code MQRC_NO_MSG_AVAILABLE.

On MVS/ESA, the period of time that the MQGET call actually waits is affected by system loading and work-scheduling considerations, and can vary between the value specified for *WaitInterval* and approximately 250 milliseconds greater than *WaitInterval*.

WaitInterval is used in conjunction with the MQGMO_WAIT or MQGMO_SET_SIGNAL option. It is ignored if neither of these is specified. If one of these is specified, *WaitInterval* must be greater than or equal to zero, or the following special value:

MQWI_UNLIMITED
Unlimited wait interval.

The initial value of this field is `0`.

Signal1 (MQLONG)
Signal.

This is an input field that is used only in conjunction with the MQGMO_SET_SIGNAL option; it identifies a signal that is to be delivered when a message is available. The data type and usage of this field are determined by the environment; for this reason, signals should not be used by applications which are intended to be portable between different environments.

- On MVS/ESA, this field contains the address of an Event Control Block (ECB). The ECB must be cleared by the application before the MQGET call is issued. The storage containing the ECB must not be freed until the queue is closed. The ECB is posted by the queue manager with one of the signal completion codes described below. These completion codes are set in bits 2 through 31 of the ECB—the area defined in the MVS mapping macro IHAECB as being for a user completion code.
- On 32-bit Windows, this field contains the window handle of a window to which the signal is sent. If this is zero, the signal is sent to the

thread requesting the signal. The signal is a Windows message with the identifier specified by the *Signal2* field. The message contains a signal completion code in the WPARAM field.

- In all other environments, this is a reserved field; its value is not significant.

The signal completion codes are:

MQEC_MSG_ARRIVED
: Message has arrived.

  A suitable message has arrived on the queue. This message has not been reserved for the caller; a second MQGET request must be issued, but note that another application might retrieve the message before the second request is made.

MQEC_WAIT_INTERVAL_EXPIRED
: Requested wait period has expired.

  The specified *WaitInterval* has expired without a suitable message arriving.

MQEC_WAIT_CANCELED
: Requested wait period has been canceled.

  The wait was canceled for an indeterminate reason (such as the queue manager terminating, or the queue being disabled). The request must be reissued if further diagnosis is required.

MQEC_Q_MGR_QUIESCING
: Queue manager quiescing.

  The wait was canceled because the queue manager has entered the quiescing state (MQGMO_FAIL_IF_QUIESCING was specified on the MQGET call).

MQEC_CONNECTION_QUIESCING
: Connection quiescing.

  The wait was canceled because the connection has entered the quiescing state (MQGMO_FAIL_IF_QUIESCING was specified on the MQGET call).

The initial value of this field is determined by the environment:

- On MVS/ESA, the initial value is the null pointer.
- In all other environments, the initial value is `0`.

Signal2 (MQLONG)
: Signal identifier.

  This is an input field that is used only in conjunction with the MQGMO_SET_SIGNAL option. The data type and usage of this field are determined by the environment:

  - On 32-bit Windows, this field contains the identifier of a Windows message that is sent to the application window (as specified by the *Signal1* field) to signal that a suitable message has arrived. The Windows call `RegisterWindowMessage` should be used to obtain a suitable identifier.

- In all other environments, this is a reserved field; its value is not significant.

The initial value of this field is 0.

ResolvedQName (MQCHAR48)
Resolved name of 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 local queue manager. This will be different from the name used to open the queue if:

- An alias queue was opened (in which case, the name of the local queue to which the alias resolved is returned), or
- A model queue was opened (in which case, the name of the dynamic local queue is returned).

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

The remaining fields in this structure are not present if *Version* is less than MQGMO_VERSION_2.

MatchOptions (MQLONG)
Options controlling selection criteria used for MQGET.

These options allow the application to choose which fields in the *MsgDesc* parameter will be used to select the message returned by the MQGET call. The application sets the required options in this field, and then sets the corresponding fields in the *MsgDesc* parameter to the values required for those fields. Only messages that have those values in the MQMD for the message are candidates for retrieval using that *MsgDesc* parameter on the MQGET call. Fields for which the corresponding match option is *not* specified are ignored when selecting the message to be returned. If no selection criteria are to be used on the MQGET call (that is, *any* message is acceptable), *MatchOptions* should be set to MQMO_NONE.

If MQGMO_LOGICAL_ORDER is specified, only certain messages are eligible for return by the next MQGET call:

- If there is no current group or logical message, only messages that have *MsgSeqNumber* equal to 1 and *Offset* equal to 0 are eligible for return. In this situation, one or more of the following match options can be used to select which of the eligible messages is the one actually returned:

  MQMO_MATCH_MSG_ID
  MQMO_MATCH_CORREL_ID
  MQMO_MATCH_GROUP_ID

- If there *is* a current group or logical message, only the next message in the group or next segment in the logical message is eligible for return, and this cannot be altered by specifying MQMO_★ options.

In both of the above cases, match options which are not applicable can still be specified, but the value of the relevant field in the *MsgDesc* parameter must match the value of the corresponding field in the message

to be returned; the call fails with reason code MQRC_MATCH_OPTIONS_ERROR is this condition is not satisfied.

MatchOptions is ignored if either MQGMO_MSG_UNDER_CURSOR or MQGMO_BROWSE_MSG_UNDER_CURSOR is specified.

One or more of the following match options can be specified:

MQMO_MATCH_MSG_ID
: Retrieve message with specified message identifier.

  This option specifies that the message to be retrieved must have a message identifier that matches the value of the *MsgId* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that may apply (for example, the correlation identifier).

  If this option is not specified, the *MsgId* field in the *MsgDesc* parameter is ignored, and any message identifier will match.

  Note: The message identifier MQMI_NONE is a special value that matches *any* message identifier in the MQMD for the message. Therefore, specifying MQMO_MATCH_MSG_ID with MQMI_NONE is the same as *not* specifying MQMO_MATCH_MSG_ID.

MQMO_MATCH_CORREL_ID
: Retrieve message with specified correlation identifier.

  This option specifies that the message to be retrieved must have a correlation identifier that matches the value of the *CorrelId* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that may apply (for example, the message identifier).

  If this option is not specified, the *CorrelId* field in the *MsgDesc* parameter is ignored, and any correlation identifier will match.

  Note: The correlation identifier MQCI_NONE is a special value that matches *any* correlation identifier in the MQMD for the message. Therefore, specifying MQMO_MATCH_CORREL_ID with MQCI_NONE is the same as *not* specifying MQMO_MATCH_CORREL_ID.

MQMO_MATCH_GROUP_ID
: Retrieve message with specified group identifier.

  This option specifies that the message to be retrieved must have a group identifier that matches the value of the *GroupId* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that may apply (for example, the correlation identifier).

  If this option is not specified, the *GroupId* field in the *MsgDesc* parameter is ignored, and any group identifier will match.

  Note: The group identifier MQGI_NONE is a special value that matches *any* group identifier in the MQMD for the message. Therefore, specifying MQMO_MATCH_GROUP_ID with MQGI_NONE is the same as *not* specifying MQMO_MATCH_GROUP_ID.

MQMO_MATCH_MSG_SEQ_NUMBER
: Retrieve message with specified message sequence number.

  This option specifies that the message to be retrieved must have a message sequence number that matches the value of the *MsgSeqNumber* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that may apply (for example, the group identifier).

  If this option is not specified, the *MsgSeqNumber* field in the *MsgDesc* parameter is ignored, and any message sequence number will match.

MQMO_MATCH_OFFSET
: Retrieve message with specified offset.

  This option specifies that the message to be retrieved must have an offset that matches the value of the *Offset* field in the *MsgDesc* parameter of the MQGET call. This match is in addition to any other matches that may apply (for example, the message sequence number).

  If this option is not specified, the *Offset* field in the *MsgDesc* parameter is ignored, and any offset will match.

If none of the options described above is specified, the following option can be used:

MQMO_NONE
: No matches.

  This option specifies that no matches are to be used in selecting the message to be returned; therefore, all messages on the queue are eligible for retrieval (but subject to control by the MQGMO_ALL_MSGS_AVAILABLE, MQGMO_ALL_SEGMENTS_AVAILABLE, and MQGMO_COMPLETE_MSG options).

  MQMO_NONE is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of this field is MQMO_MATCH_MSG_ID with MQMO_MATCH_CORREL_ID. This field is not present if *Version* is less than MQGMO_VERSION_2.

Note: The initial value of the *MatchOptions* field is defined for compatibility with earlier MQSeries queue managers. However, when reading a series of messages from a queue without using selection criteria, this initial value requires the application to reset the *MsgId* and *CorrelId* fields to MQMI_NONE and MQCI_NONE prior to each MQGET call. The need to reset *MsgId* and *CorrelId* can be avoided by setting *Version* to MQGMO_VERSION_2, and *MatchOptions* to MQMO_NONE.

GroupStatus (MQCHAR)
: Flag indicating whether message retrieved is in a group.

  It has one of the following values:

MQGS_NOT_IN_GROUP
Message is not in a group.

MQGS_MSG_IN_GROUP
Message is in a group, but is not the last in the group.

MQGS_LAST_MSG_IN_GROUP
Message is the last in the group.

This is also the value returned if the group consists of only one message.

This is an output field. The initial value of this field is MQGS_NOT_IN_GROUP. This field is not present if *Version* is less than MQGMO_VERSION_2.

SegmentStatus (MQCHAR)
Flag indicating whether message retrieved is a segment of a logical message.

It has one of the following values:

MQSS_NOT_A_SEGMENT
Message is not a segment.

MQSS_SEGMENT
Message is a segment, but is not the last segment of the logical message.

MQSS_LAST_SEGMENT
Message is the last segment of the logical message.

This is also the value returned if the logical message consists of only one segment.

This is an output field. The initial value of this field is MQSS_NOT_A_SEGMENT. This field is not present if *Version* is less than MQGMO_VERSION_2.

Segmentation (MQCHAR)
Flag indicating whether further segmentation is allowed for the message retrieved.

It has one of the following values:

MQSEG_INHIBITED
Segmentation not allowed.

MQSEG_ALLOWED
Segmentation allowed.

This is an output field. The initial value of this field is MQSEG_INHIBITED. This field is not present if *Version* is less than MQGMO_VERSION_2.

Reserved1 (MQCHAR)
Reserved.

This is a reserved field. The initial value of this field is a blank character. This field is not present if *Version* is less than MQGMO_VERSION_2.

Table 31. Initial values of fields in MQGMO

Field name	Name of constant	Value of constant
StrucId	MQGMO_STRUC_ID	'GMOb' (See note 1)
Version	MQGMO_VERSION_1	1
Options	MQGMO_NO_WAIT	0
WaitInterval	None	0
Signal1	None	Null pointer on MVS/ESA; 0 otherwise
Signal2	None	0
ResolvedQName	None	Blanks (See note 2)
MatchOptions	MQMO_MATCH_MSG_ID + MQMO_MATCH_CORREL_ID	3
GroupStatus	MQGS_NOT_IN_GROUP	'b'
SegmentStatus	MQSS_NOT_A_SEGMENT	'b'
Segmentation	MQSEG_INHIBITED	'b'
Reserved1	None	'b'

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQGMO_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQGMO MyGMO = {MQGMO_DEFAULT};
```

C language declaration

```
typedef struct tagMQGMO {
  MQCHAR4   StrucId;        /* Structure identifier */
  MQLONG    Version;        /* Structure version number */
  MQLONG    Options;        /* Options that control the action of
                               MQGET */
  MQLONG    WaitInterval;   /* Wait interval */
  MQLONG    Signal1;        /* Signal */
  MQLONG    Signal2;        /* Signal identifier */
  MQCHAR48  ResolvedQName;  /* Resolved name of destination queue */
  MQLONG    MatchOptions;   /* Options controlling selection criteria
                               used for MQGET */
  MQCHAR    GroupStatus;    /* Flag indicating whether message retrieved
                               is in a group */
  MQCHAR    SegmentStatus;  /* Flag indicating whether message retrieved
                               is a segment of a logical message */
  MQCHAR    Segmentation;   /* Flag indicating whether segmentation is
                               allowed for the message retrieved */
  MQCHAR    Reserved1;      /* Reserved */
 } MQGMO;
```

COBOL language declaration

```
**   MQGMO structure
  10 MQGMO.
**    Structure identifier
   15 MQGMO-STRUCID       PIC X(4).
**    Structure version number
   15 MQGMO-VERSION       PIC S9(9) BINARY.
**    Options that control the action of MQGET
   15 MQGMO-OPTIONS       PIC S9(9) BINARY.
**    Wait interval
   15 MQGMO-WAITINTERVAL  PIC S9(9) BINARY.
**    Signal
   15 MQGMO-SIGNAL1       PIC S9(9) BINARY.
**    Reserved
   15 MQGMO-SIGNAL2       PIC S9(9) BINARY.
**    Resolved name of destination queue
   15 MQGMO-RESOLVEDQNAME PIC X(48).
**    Options controlling selection criteria used for MQGET
   15 MQGMO-MATCHOPTIONS  PIC S9(9) BINARY.
**    Flag indicating whether message retrieved is in a group
   15 MQGMO-GROUPSTATUS   PIC X.
**    Flag indicating whether message retrieved is a segment of a
**    logical message
   15 MQGMO-SEGMENTSTATUS PIC X.
**    Flag indicating whether segmentation is allowed for the
**    message retrieved
   15 MQGMO-SEGMENTATION  PIC X.
**    Reserved
   15 MQGMO-RESERVED1     PIC X.
```

PL/I declaration (AIX, MVS/ESA, OS/2, and Windows NT)

```
dcl
 1 MQGMO based,
  3 StrucId       char(4),       /* Structure identifier */
  3 Version       fixed bin(31), /* Structure version number */
  3 Options       fixed bin(31), /* Options that control the action of
                                    MQGET */
  3 WaitInterval  fixed bin(31), /* Wait interval */
  3 Signal1       fixed bin(31), /* Signal */
  3 Signal2       fixed bin(31), /* Reserved */
  3 ResolvedQName char(48),      /* Resolved name of destination
                                    queue */
  3 MatchOptions  fixed bin(31), /* Options controlling selection cri-
                                    teria used for MQGET */
  3 GroupStatus   char(1),       /* Flag indicating whether message
                                    retrieved is in a group */
  3 SegmentStatus char(1),       /* Flag indicating whether message
                                    retrieved is a segment of a logical
                                    message */
  3 Segmentation  char(1),       /* Flag indicating whether segmentation
                                    is allowed for the message
                                    retrieved */
  3 Reserved1     char(1);       /* Reserved */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQGMO                    DSECT
MQGMO_STRUCID            DS   CL4      Structure identifier
MQGMO_VERSION            DS   F        Structure version number
MQGMO_OPTIONS            DS   F        Options that control the
*                                      action of MQGET
MQGMO_WAITINTERVAL       DS   F        Wait interval
MQGMO_SIGNAL1            DS   F        Signal
MQGMO_SIGNAL2            DS   F        Reserved
MQGMO_RESOLVEDQNAME      DS   CL48     Resolved name of destination
*                                      queue
MQGMO_LENGTH             EQU  *-MQGMO  Length of structure
                         ORG  MQGMO
MQGMO_AREA               DS   CL(MQGMO_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQGMO^DEF (*);
BEGIN
     STRUCT       STRUCID;
BEGIN STRING BYTE [0:3]; END;
INT(32)        VERSION;
INT(32)        OPTIONS;
INT(32)        WAITINTERVAL;
INT(32)        SIGNAL1;
INT(32)        SIGNAL2;
STRUCT         RESOLVEDQNAME;
BEGIN STRING BYTE [0:47]; END;
 END;
```

MQIIH – IMS bridge header

The following table summarizes the fields in the structure.

Table 32. Fields in MQIIH

Field	Description	Page
StrucId	Structure identifier	92
Version	Structure version number	92
StrucLength	Length of MQIIH structure	92
Format	MQ format name	92
LTermOverride	Logical terminal override	93
MFSMapName	Message format services map name	93
ReplyToFormat	MQ format name of reply message	93
Authenticator	RACF password or passticket	93
TranInstanceId	Transaction instance identifier	94
TranState	Transaction state	94
CommitMode	Commit mode	94
SecurityScope	Security scope	94

The MQIIH structure describes the information that must be present at the start of a message sent to the IMS bridge through MQSeries for MVS/ESA. The format name of this structure is MQFMT_IMS.

Special conditions apply to the character set and encoding used for the MQIIH structure and application message data:

- Applications that connect to the queue manager which owns the IMS bridge queue must provide an MQIIH structure that is in the character set and encoding of the queue manager. This is because data conversion of the MQIIH structure is not performed in this case.
- Applications that connect to other queue managers can provide an MQIIH structure that is in any of the supported character sets and encodings; conversion of the MQIIH and application message data is performed by the queue manager as necessary.

  Note: There is one exception to this. If the queue manager which owns the IMS bridge queue is using CICS for distributed queuing, the MQIIH must be in the character set and encoding of that queue manager.
- The application message data following the MQIIH structure must be in the same character set and encoding as the MQIIH structure. The *CodedCharSetId* and *Encoding* fields in the MQIIH structure cannot be used to specify the character set and encoding of the application message data.

This structure is not supported in the following environments: 16-bit Windows, 32-bit Windows.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQIIH_STRUC_ID
Identifier for IMS information header structure.

For the C programming language, the constant MQIIH_STRUC_ID_ARRAY is also defined; this has the same value as MQIIH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQIIH_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQIIH_VERSION_1
Version number for IMS information header structure.

The following constant specifies the version number of the current version:

MQIIH_CURRENT_VERSION
Current version of IMS information header structure.

The initial value of this field is MQIIH_VERSION_1.

StrucLength (MQLONG)
Length of MQIIH structure.

The value must be:

MQIIH_LENGTH_1
Length of IMS information header structure.

The initial value of this field is MQIIH_LENGTH_1.

Encoding (MQLONG)
Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

CodedCharSetId (MQLONG)
Reserved.

This is a reserved field; its value is not significant. The initial value of this field is 0.

Format (MQCHAR8)
MQ format name.

This is the MQ format name of the application message data which follows the MQIIH structure. The rules for coding this are the same as those for the *Format* field in MQMD.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

Flags (MQLONG)
Reserved.

The value must be:

MQIIH_NONE
No flags.

The initial value of this field is MQIIH_NONE.

LTermOverride (MQCHAR8)
Logical terminal override.

This is placed in the IO PCB field. It is optional; if it is not specified the TPIPE name is used. It is ignored if the first byte is blank, or null.

The length of this field is given by MQ_LTERM_OVERRIDE_LENGTH. The initial value of this field is 8 blank characters.

MFSMapName (MQCHAR8)
Message format services map name.

This is placed in the IO PCB field. It is optional. On input it represents the MID, on output it represents the MOD. It is ignored if the first byte is blank or null.

The length of this field is given by MQ_MFS_MAP_NAME_LENGTH. The initial value of this field is 8 blank characters.

ReplyToFormat (MQCHAR8)
MQ format name of reply message.

This is the MQ format name of the reply message which will be sent in response to the current message. The rules for coding this are the same as those for the *Format* field in MQMD.

The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

Authenticator (MQCHAR8)
RACF password or passticket.

This is optional; if specified, it is used with the user ID in the MQMD security context to build a Utoken that is sent to IMS to provide a security context. If it is not specified, the user ID is used without verification. This depends on the setting of the RACF switches, which may require an authenticator to be present.

This is ignored if the first byte is blank or null. The following special value may be used:

MQIAUT_NONE
No authentication.

For the C programming language, the constant MQIAUT_NONE_ARRAY is also defined; this has the same value as MQIAUT_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_AUTHENTICATOR_LENGTH. The initial value of this field is MQIAUT_NONE.

TranInstanceId (MQBYTE16)
Transaction instance identifier.

This field is used by output messages from IMS so is ignored on first input. If *TranState* is set to MQITS_IN_CONVERSATION, this must be provided in the next input, and all subsequent inputs, to enable IMS to correlate the messages to the correct conversation. The following special value may be used:

MQITII_NONE
No transaction instance id.

For the C programming language, the constant MQITII_NONE_ARRAY is also defined; this has the same value as MQITII_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_TRAN_INSTANCE_ID_LENGTH. The initial value of this field is MQITII_NONE.

TranState (MQCHAR)
Transaction state.

This indicates the IMS conversation state. This is ignored on first input because no conversation exists. On subsequent inputs it indicates whether a conversation is active or not. On output it is set by IMS. The value must be one of the following:

MQITS_IN_CONVERSATION
In conversation.

MQITS_NOT_IN_CONVERSATION
Not in conversation.

The initial value of this field is MQITS_NOT_IN_CONVERSATION.

CommitMode (MQCHAR)
Commit mode.

See the *OTMA User's Guide* for more information about IMS commit modes. The value must be one of the following:

MQICM_COMMIT_THEN_SEND
Commit then send.

This mode implies double queuing of output, but shorter region occupancy times. Fast-path and conversational transactions cannot run with this mode.

MQICM_SEND_THEN_COMMIT
Send then commit.

The initial value of this field is MQICM_COMMIT_THEN_SEND.

SecurityScope (MQCHAR)
Security scope.

This indicates the desired IMS security processing. The value must be one of the following:

MQISS_CHECK
Check security scope.

An ACEE is built in the control region, but not in the dependent region.

MQISS_FULL
Full security scope.

A cached ACEE is built in the control region and a non-cached ACEE is built in the dependent region. If you use MQISS_FULL, you must ensure that the user ID for which the ACEE is built has access to the resources used in the dependent region.

The initial value of this field is MQISS_CHECK.

Reserved (MQCHAR)
Reserved.

This is a reserved field; it must be blank.

Table 33. Initial values of fields in MQIIH

Field name	Name of constant	Value of constant
StrucId	MQIIH_STRUC_ID	'IIHb' (See note 1)
Version	MQIIH_VERSION_1	1
StrucLength	MQIIH_LENGTH_1	84
Encoding	None	0
CodedCharSetId	None	0
Format	MQFMT_NONE	'bbbbbbbb'
Flags	MQIIH_NONE	0
LTermOverride	None	'bbbbbbbb'
MFSMapName	None	'bbbbbbbb'
ReplyToFormat	MQFMT_NONE	'bbbbbbbb'
Authenticator	MQIAUT_NONE	'bbbbbbbb'
TranInstanceId	MQITII_NONE	Nulls
TranState	MQITS_NOT_IN_CONVERSATION	'b'
CommitMode	MQICM_COMMIT_THEN_SEND	'0'
SecurityScope	MQISS_CHECK	'C'
Reserved	None	'b'

Notes:

1. The symbol 'b' represents a single blank character.
2. In the C programming language, the macro variable MQIIH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQIIH MyIIH = {MQIIH_DEFAULT};
```

C language declaration

```
typedef struct tagMQIIH {
  MQCHAR4   StrucId;         /* Structure identifier */
  MQLONG    Version;         /* Structure version number */
  MQLONG    StrucLength;     /* Length of MQIIH structure */
  MQLONG    Encoding;        /* Reserved */
  MQLONG    CodedCharSetId;  /* Reserved */
  MQCHAR8   Format;          /* Format name */
  MQLONG    Flags;           /* Reserved */
  MQCHAR8   LTermOverride;   /* Logical terminal override */
  MQCHAR8   MFSMapName;      /* Message format services map name */
  MQCHAR8   ReplyToFormat;   /* Format name of reply message */
  MQCHAR8   Authenticator;   /* RACF password or passticket */
  MQBYTE16  TranInstanceId;  /* Transaction instance id */
  MQCHAR    TranState;       /* Transaction state */
  MQCHAR    CommitMode;      /* Commit mode */
  MQCHAR    SecurityScope;   /* Security scope */
  MQCHAR    Reserved;        /* Reserved */
 } MQIIH;
```

COBOL language declaration

```
**   MQIIH structure
  10 MQIIH.
**    Structure identifier
   15 MQIIH-STRUCID        PIC X(4).
**    Structure version number
   15 MQIIH-VERSION        PIC S9(9) BINARY.
**    Length of MQIIH structure
   15 MQIIH-STRUCLENGTH    PIC S9(9) BINARY.
**    Reserved
   15 MQIIH-ENCODING       PIC S9(9) BINARY.
**    Reserved
   15 MQIIH-CODEDCHARSETID PIC S9(9) BINARY.
**    Format name
   15 MQIIH-FORMAT         PIC X(8).
**    Reserved
   15 MQIIH-FLAGS          PIC S9(9) BINARY.
**    Logical terminal override
   15 MQIIH-LTERMOVERRIDE  PIC X(8).
**    Message format services map name
   15 MQIIH-MFSMAPNAME     PIC X(8).
**    Format name of reply message
   15 MQIIH-REPLYTOFORMAT  PIC X(8).
**    RACF password or passticket
   15 MQIIH-AUTHENTICATOR  PIC X(8).
**    Transaction instance id
   15 MQIIH-TRANINSTANCEID PIC X(16).
**    Transaction state
   15 MQIIH-TRANSTATE      PIC X.
**    Commit mode
   15 MQIIH-COMMITMODE     PIC X.
**    Security scope
   15 MQIIH-SECURITYSCOPE  PIC X.
**    Reserved
   15 MQIIH-RESERVED       PIC X.
```

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

```
dcl
 1 MQIIH based,
  3 StrucId        char(4),       /* Structure identifier */
  3 Version        fixed bin(31), /* Structure version number */
  3 StrucLength    fixed bin(31), /* Length of MQIIH structure */
  3 Encoding       fixed bin(31), /* Reserved */
  3 CodedCharSetId fixed bin(31), /* Reserved */
  3 Format         char(8),       /* Format name */
  3 Flags          fixed bin(31), /* Reserved */
  3 LTermOverride  char(8),       /* Logical terminal override */
  3 MFSMapName     char(8),       /* Message format services map name */
  3 ReplyToFormat  char(8),       /* Format name of reply message */
  3 Authenticator  char(8),       /* RACF password or passticket */
  3 TranInstanceId char(16),      /* Transaction instance id */
  3 TranState      char(1),       /* Transaction state */
  3 CommitMode     char(1),       /* Commit mode */
  3 SecurityScope  char(1),       /* Security scope */
  3 Reserved       char(1);       /* Reserved */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQIIH                   DSECT
MQIIH_STRUCID           DS   CL4      Structure identifier
MQIIH_VERSION           DS   F        Structure version number
MQIIH_STRUCLENGTH       DS   F        Length of MQIIH structure
MQIIH_ENCODING          DS   F        Reserved
MQIIH_CODEDCHARSETID    DS   F        Reserved
MQIIH_FORMAT            DS   CL8      Format name
MQIIH_FLAGS             DS   F        Reserved
MQIIH_LTERMOVERRIDE     DS   CL8      Logical terminal override
MQIIH_MFSMAPNAME        DS   CL8      Message format services map
*                                     name
MQIIH_REPLYTOFORMAT     DS   CL8      Format name of reply message
MQIIH_AUTHENTICATOR     DS   CL8      RACF password or passticket
MQIIH_TRANINSTANCEID    DS   XL16     Transaction instance id
MQIIH_TRANSTATE         DS   CL1      Transaction state
MQIIH_COMMITMODE        DS   CL1      Commit mode
MQIIH_SECURITYSCOPE     DS   CL1      Security scope
MQIIH_RESERVED          DS   CL1      Reserved
MQIIH_LENGTH            EQU  *-MQIIH  Length of structure
                        ORG  MQIIH
MQIIH_AREA              DS   CL(MQIIH_LENGTH)
```

MQMD – Message descriptor

The following table summarizes the fields in the structure.

Table 34. Fields in MQMD

Field	Description	Page
StrucId	Structure identifier	100
Version	Structure version number	100
Report	Options for report messages	101
MsgType	Message type	112
Expiry	Message lifetime	113
Feedback	Feedback or reason code	115
Encoding	Data encoding	118
CodedCharSetId	Coded character set identifier	118
Format	Format name	119
Priority	Message priority	124
Persistence	Message persistence	125
MsgId	Message identifier	126
CorrelId	Correlation identifier	128
BackoutCount	Backout counter	129
ReplyToQ	Name of reply queue	130
ReplyToQMgr	Name of reply queue manager	131
UserIdentifier	User identifier	131
AccountingToken	Accounting token	133
ApplIdentityData	Application data relating to identity	134
PutApplType	Type of application that put the message	134
PutApplName	Name of application that put the message	136
PutDate	Date when message was put	137
PutTime	Time when message was put	138
ApplOriginData	Application data relating to origin	139
Note: The remaining fields are supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.		
GroupId	Group identifier	139
MsgSeqNumber	Sequence number of logical message within group	141
Offset	Offset of data in physical message from start of logical message.	141
MsgFlags	Message flags	142
OriginalLength	Length of original message	147

The MQMD structure contains the control information that accompanies the application data when a message travels between the sending and receiving applications.

Character data in the message descriptor is in the character set of the queue manager to which the application is connected; this is given by the *CodedCharSetId* queue-manager attribute. Numeric data in the message descriptor is in the native machine encoding (given by MQENC_NATIVE).

If the sending and receiving queue managers use different character sets or encodings, the data in the message descriptor is converted automatically—it is not necessary for the receiving application to perform these conversions.

If the application message data requires conversion, this can be accomplished by means of a user-written exit invoked when the message is retrieved using the MQGET call. For further information, see:

- The MQGMO_CONVERT option described in "MQGMO – Get-message options" on page 56
- The usage note describing MQGMO_CONVERT in "MQGET – Get message" on page 273
- *MQSeries Application Programming Guide*

When a message is on a transmission queue, some of the fields in MQMD are set to particular values; see "MQXQH – Transmission queue header" on page 227 for details.

The current version of MQMD is MQMD_VERSION_2. Fields that exist only in the version-2 structure are identified as such in the descriptions that follow. The declarations of MQMD provided in the header, COPY, and INCLUDE files for the supported programming languages contain the new fields, but the initial value provided for the *Version* field is MQMD_VERSION_1; this ensures compatibility with existing applications. To use the new fields, the application must set the version number to MQMD_VERSION_2. A declaration for the version-1 structure is available with the name MQMD1. Applications which are intended to be portable between several environments should use a version-2 MQMD only if all of those environments support version 2.

The version-2 structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

A version-2 MQMD is generally equivalent to using a version-1 MQMD and prefixing the application message data with an MQMDE structure. However, if all of the fields in the MQMDE structure have their default values, the MQMDE can be omitted. A version-1 MQMD plus MQMDE are used as follows:

- On the MQPUT and MQPUT1 calls, if the application provides a version-1 MQMD, the application can optionally prefix the message data with an MQMDE, setting the *Format* field in MQMD to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. If the application does not provide an MQMDE, the queue manager assumes default values for the fields in the MQMDE.

  Note: Several of the fields that exist in the version-2 MQMD but not the version-1 MQMD are input/output fields on MQPUT and MQPUT1. However, the queue manager *does not* return any values in the equivalent fields in the MQMDE on output from the MQPUT and MQPUT1 calls; if the application requires those output values, it must use a version-2 MQMD.

- On the MQGET call, if the application provides a version-1 MQMD, the queue manager prefixes the message returned with an MQMDE, but only if one or more of the fields in the MQMDE has a non-default value. The *Format* field in MQMD will have the value MQFMT_MD_EXTENSION to indicate that an MQMDE is present.

The default values that the queue manager used for the fields in the MQMDE are the same as the initial values of those fields, shown in Table 38 on page 158.

This structure is an input/output parameter for the MQGET, MQPUT, and MQPUT1 calls.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQMD_STRUC_ID
Identifier for message descriptor structure.

For the C programming language, the constant MQMD_STRUC_ID_ARRAY is also defined; this has the same value as MQMD_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQMD_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be one of the following:

MQMD_VERSION_1
Version-1 message descriptor structure.

This version is supported in all environments.

MQMD_VERSION_2
Version-2 message descriptor structure.

This version is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields that exist only in the version-2 structure are identified as such in the descriptions that follow.

Note: When a version-2 MQMD is used, the queue manager performs additional checks on any MQ header structures that may be present at the beginning of the application message data; for further details see usage note 3 on page 320 for the MQPUT call.

The following constant specifies the version number of the current version:

MQMD_CURRENT_VERSION
Current version of message descriptor structure.

This is always an input field. The initial value of this field is MQMD_VERSION_1.

Report (MQLONG)
Options for report messages.

A report is a message about another message, used to inform an application about expected or unexpected events that relate to the original message. The *Report* field enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and also (for both reports and replies) how the message and correlation identifiers in the report or reply message are to be set. Any or all (or none) of the following report types can be requested:

- Exception
- Expiration
- Confirm on arrival (COA)
- Confirm on delivery (COD)
- Positive action notification (PAN)
- Negative action notification (NAN)

If more than one type of report message is required, or other report options are needed, the values can be:

- Added together (do not add the same constant more than once), or
- Combined using the bitwise OR operation (if the programming language supports bit operations).

The application that receives the report message can determine the reason the report was generated by examining the *Feedback* field in the MQMD; see the *Feedback* field for more details.

Exception options: You can specify one of the following to request an exception report message:

MQRO_EXCEPTION
Exception reports required.

This type of report can be generated by a message channel agent when a message is sent to another queue manager and the message cannot be delivered to the specified destination queue. For example, the destination queue or an intermediate transmission queue might be full, or the message might be too big for the queue.

Generation of the exception report message depends on the persistence of the original message, and the speed of the message channel (normal or fast) through which the original message travels:

- For all persistent messages, and for nonpersistent messages traveling through normal message channels, the exception report is generated *only* if the action specified by the sending application for the error condition can be completed successfully. The sending application can specify one of the following actions to control the disposition of the original message when the error condition arises:
  - MQRO_DEAD_LETTER_Q (this causes the original message to be placed on the dead-letter queue).

– MQRO_DISCARD_MSG (this causes the original message to be discarded).

If the action specified by the sending application cannot be completed successfully, the original message is left on the transmission queue, and no exception report message is generated.

- For nonpersistent messages traveling through fast message channels, the original message is removed from the transmission queue and the exception report generated *even if* the specified action for the error condition cannot be completed successfully. For example, if MQRO_DEAD_LETTER_Q is specified, but the original message cannot be placed on the dead-letter queue because (say) that queue is full, the exception report message is generated and the original message discarded.

  Refer to the *MQSeries Intercommunication* book for more information about normal and fast message channels.

An exception report is not generated if the application that put the original message can be notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call.

Applications can also send exception reports, to indicate that a message that it has received cannot be processed (for example, because it is a debit transaction that would cause the account to exceed its credit limit).

Message data from the original message is not included with the report message.

Do not specify more than one of MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA, and MQRO_EXCEPTION_WITH_FULL_DATA.

MQRO_EXCEPTION_WITH_DATA
Exception reports with data required.

This is the same as MQRO_EXCEPTION, except that the first 100 bytes of the application message data from the original message are included in the report message. If the length of the message data in the original message is less than 100 bytes, the length of the message data in the report is the same length as the original message.

Do not specify more than one of MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA, and MQRO_EXCEPTION_WITH_FULL_DATA.

MQRO_EXCEPTION_WITH_FULL_DATA
Exception reports with full data required.

This is the same as MQRO_EXCEPTION, except that all of the application message data from the original message is included in the report message.

Do not specify more than one of MQRO_EXCEPTION, MQRO_EXCEPTION_WITH_DATA, and MQRO_EXCEPTION_WITH_FULL_DATA.

On MVS/ESA, the MQRO_EXCEPTION_WITH_FULL_DATA option is not supported.

Expiration options: You can specify one of the following to request an expiration report message:

MQRO_EXPIRATION
Expiration reports required.

This type of report is generated by the queue manager if the message is discarded prior to delivery to an application because its expiry time has passed (see the *Expiry* field). If this option is not set, no report message is generated if a message is discarded for this reason (even if one of the MQRO_EXCEPTION_★ options is specified).

Message data from the original message is not included with the report message.

Do not specify more than one of MQRO_EXPIRATION, MQRO_EXPIRATION_WITH_DATA, and MQRO_EXPIRATION_WITH_FULL_DATA.

MQRO_EXPIRATION_WITH_DATA
Expiration reports with data required.

This is the same as MQRO_EXPIRATION, except that the first 100 bytes of the application message data from the original message are included in the report message. If the length of the message data in the original message is less than 100 bytes, the length of the message data in the report is the same length as the original message.

Do not specify more than one of MQRO_EXPIRATION, MQRO_EXPIRATION_WITH_DATA, and MQRO_EXPIRATION_WITH_FULL_DATA.

MQRO_EXPIRATION_WITH_FULL_DATA
Expiration reports with full data required.

This is the same as MQRO_EXPIRATION, except that all of the application message data from the original message is included in the report message.

Do not specify more than one of MQRO_EXPIRATION, MQRO_EXPIRATION_WITH_DATA, and MQRO_EXPIRATION_WITH_FULL_DATA.

On MVS/ESA, the MQRO_EXPIRATION_WITH_FULL_DATA option is not supported.

Confirm-on-arrival options: You can specify one of the following to request a confirm-on-arrival report message:

MQRO_COA
Confirm-on-arrival reports required.

This type of report is generated by the queue manager that owns the destination queue, when the message is placed on the destination queue. Message data from the original message is not included with the report message.

If the message is put as part of a unit of work, and the destination queue is a local queue, the COA report message generated by the queue manager becomes available for retrieval only if and when the unit of work is committed.

A COA report is not generated if the *Format* field in the message descriptor is MQFMT_XMIT_Q_HEADER or MQFMT_DEAD_LETTER_HEADER. This prevents a COA report being generated if the message is put on a transmission queue, or is undeliverable and put on a dead-letter queue.

Do not specify more than one of MQRO_COA, MQRO_COA_WITH_DATA, and MQRO_COA_WITH_FULL_DATA.

MQRO_COA_WITH_DATA
Confirm-on-arrival reports with data required.

This is the same as MQRO_COA, except that the first 100 bytes of the application message data from the original message are included in the report message. If the length of the message data in the original message is less than 100 bytes, the length of the message data in the report is the same length as the original message.

Do not specify more than one of MQRO_COA, MQRO_COA_WITH_DATA, and MQRO_COA_WITH_FULL_DATA.

MQRO_COA_WITH_FULL_DATA
Confirm-on-arrival reports with full data required.

This is the same as MQRO_COA, except that all of the application message data from the original message is included in the report message.

Do not specify more than one of MQRO_COA, MQRO_COA_WITH_DATA, and MQRO_COA_WITH_FULL_DATA.

On MVS/ESA, the MQRO_COA_WITH_FULL_DATA option is not supported.

Confirm-on-delivery options: You can specify one of the following to request a confirm-on-delivery report message:

MQRO_COD
Confirm-on-delivery reports required.

This type of report is generated by the queue manager when an application retrieves the message from the destination queue in a way that causes the message to be deleted from the queue. Message data from the original message is not included with the report message.

If the message is retrieved as part of a unit of work, the report message is generated within the same unit of work, so that the report is not available until the unit of work is committed. If the unit of work is backed out, the report is not sent.

A COD report is not always generated if a message is retrieved with the MQGMO_MARK_SKIP_BACKOUT option. If the primary unit of work is backed out but the secondary unit of work is committed, the message is removed from the queue, but a COD report is not generated.

A COD report is not generated if the *Format* field in the message descriptor is MQFMT_DEAD_LETTER_HEADER. This prevents a COD report being generated if the message is undeliverable and put on a dead-letter queue.

MQRO_COD is not valid if the destination queue is an XCF queue.

Do not specify more than one of MQRO_COD, MQRO_COD_WITH_DATA, and MQRO_COD_WITH_FULL_DATA.

MQRO_COD_WITH_DATA
Confirm-on-delivery reports with data required.

This is the same as MQRO_COD, except that the first 100 bytes of the application message data from the original message are included in the report message. If the length of the message data in the original message is less than 100 bytes, the length of the message data in the report is the same length as the original message.

Note that any truncation of the original message on retrieval (using the MQGMO_ACCEPT_TRUNCATED_MSG option) has no effect on the size of the message data in the COD report.

MQRO_COD_WITH_DATA is not valid if the destination queue is an XCF queue.

Do not specify more than one of MQRO_COD, MQRO_COD_WITH_DATA, and MQRO_COD_WITH_FULL_DATA.

MQRO_COD_WITH_FULL_DATA
Confirm-on-delivery reports with full data required.

This is the same as MQRO_COD, except that all of the application message data from the original message is included in the report message.

MQRO_COD_WITH_FULL_DATA is not valid if the destination queue is an XCF queue.

Do not specify more than one of MQRO_COD, MQRO_COD_WITH_DATA, and MQRO_COD_WITH_FULL_DATA.

On MVS/ESA, the MQRO_COD_WITH_FULL_DATA option is not supported.

Action-notification options: You can specify one or both of the following to request that the receiving application send a positive-action or negative-action report message:

MQRO_PAN
Positive action notification reports required.

This type of report is generated by the application that retrieves the message and acts upon it. It indicates that the action requested in the message has been performed successfully. The application generating the report determines whether or not any data is to be included with the report.

Other than conveying this request to the application retrieving the message, the queue manager takes no action based upon this option. It is the responsibility of the retrieving application to generate the report if appropriate.

MQRO_NAN
: Negative action notification reports required.

  This type of report is generated by the application that retrieves the message and acts upon it. It indicates that the action requested in the message has *not* been performed successfully. The application generating the report determines whether or not any data is to be included with the report. For example, it may be desirable to include some data indicating why the request could not be performed.

  Other than conveying this request to the application retrieving the message, the queue manager takes no action based upon this option. It is the responsibility of the retrieving application to generate the report if appropriate.

Determination of which conditions correspond to a positive action and which correspond to a negative action is the responsibility of the application. However, it is recommended that if the request has been only partially performed, a NAN report rather than a PAN report should be generated if requested. It is also recommended that every possible condition should correspond to either a positive action, or a negative action, but not both.

Message-identifier options: You can specify one of the following to control how the *MsgId* of the report message (or of the reply message) is to be set:

MQRO_NEW_MSG_ID
: New message identifier.

  This is the default action, and indicates that if a report or reply is generated as a result of this message, a new *MsgId* is to be generated for the report or reply message.

MQRO_PASS_MSG_ID
: Pass message identifier.

  If a report or reply is generated as a result of this message, the *MsgId* of this message is to be copied to the *MsgId* of the report or reply message.

  If this option is not specified, MQRO_NEW_MSG_ID is assumed.

Correlation-identifier options: You can specify one of the following to control how the *CorrelId* of the report message (or of the reply message) is to be set:

MQRO_COPY_MSG_ID_TO_CORREL_ID
: Copy message identifier to correlation identifier.

  This is the default action, and indicates that if a report or reply is generated as a result of this message, the *MsgId* of this message is to be copied to the *CorrelId* of the report or reply message.

MQRO_PASS_CORREL_ID
: Pass correlation identifier.

  If a report or reply is generated as a result of this message, the *CorrelId* of this message is to be copied to the *CorrelId* of the report or reply message.

If this option is not specified, MQRO_COPY_MSG_ID_TO_CORREL_ID is assumed.

Servers replying to requests or generating report messages are recommended to check whether the MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID options were set in the original message. If they were, the servers should take the action described for those options. If neither is set, the servers should take the corresponding default action.

Disposition options: You can specify one of the following to control the disposition of the original message when it cannot be delivered to the destination queue:

MQRO_DEAD_LETTER_Q
Place message on dead-letter queue.

This is the default action, and indicates that the message should be placed on the dead-letter queue, if the message cannot be delivered to the destination queue. An exception report message will be generated, if one was requested by the sender.

MQRO_DISCARD_MSG
Discard message.

This indicates that the message should be discarded if it cannot be delivered to the destination queue. An exception report message will be generated, if one was requested by the sender.

On MVS/ESA, the MQRO_DISCARD_MSG option is not supported.

If it is desired to return the original message to the sender, without the original message being placed on the dead-letter queue, the sender should specify MQRO_DISCARD_MSG with MQRO_EXCEPTION_WITH_FULL_DATA.

Default option: You can specify the following if no report options are required:

MQRO_NONE
No reports required.

This value can be used to indicate that no other options have been specified. MQRO_NONE is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

General information: All report types required must be specifically requested by the application sending the original message. For example, if a COA report is requested, but an exception report is not (with or without the data option in either case), a COA report is generated when the message is placed on the destination queue, but no exception report is generated if the destination queue is full when the message arrives there. If no *Report* options are set, no report messages are generated by the queue manager or message channel agent.

Some report options can be specified even though the local queue manager does not recognize them; this is useful when the option is to be processed by the *destination* queue manager. See Appendix C, “Report options and message flags” on page 489 for more details.

If a report message is requested, the name of the queue to which the report should be sent must be specified in the *ReplyToQ* field. When a report message is received, the nature of the report can be determined by examining the *Feedback* field in the message descriptor.

If the queue manager or message channel agent that generates a report message is unable to put the report message on the reply queue (for example, because the reply queue or transmission queue is full), the report message is placed instead on the dead-letter queue. If that *also* fails, or there is no dead-letter queue, the action taken depends on the type of the report message:

- If the report message is an exception report, the message which caused the exception report to be generated is left on its transmission queue; this ensures that the message is not lost.
- For all other report types, the report message is discarded and processing continues normally. This is done because either the original message has already been delivered safely (for COA or COD report messages), or is no longer of any interest (for an expiration report message).

Once a report message has been placed successfully on a queue (either the destination queue or an intermediate transmission queue), the message is no longer subject to special processing — it is treated just like any other message.

When the report is generated, the *ReplyToQ* queue is opened to put the report using the authority of the *UserIdentifier* of the original message, except in the following cases:

- Exception reports generated by a receiving message channel agent use the same authority as was used to put the original message.
- COA reports generated by the queue manager use the same authority as was used to put the original message.

Applications generating reports should normally use the same authority as they would have used to generate a reply; this should normally be the authority of the user ID in the original message.

If the report has to travel to a remote destination, senders and receivers can decide whether or not to accept it, in the same way as they do for other messages.

If a report message with data is requested:

- The report message is always generated with the amount of data requested by the sender of the original message. If the report message is too big for the reply queue, the processing described above occurs; the report message is never truncated in order to fit on the reply queue.
- If the *Format* of the original message is MQFMT_XMIT_Q_HEADER, the data included in the report does not include the MQXQH. The report data starts with the first byte of the data beyond the MQXQH in the original message. This occurs whether or not the queue is a transmission queue.

If a COA, COD, or expiration report message is received at the reply queue, it is guaranteed that the original message arrived, was delivered, or expired, as appropriate. However, if one or more of these report messages is requested and is *not* received, the reverse cannot be assumed, since one of the following may have occurred:

1. The report message is held up because a link is down.
2. The report message is held up because a blocking condition exists at an intermediate transmission queue or at the reply queue (for example, the queue is full or inhibited for puts).
3. The report message is on a dead-letter queue.
4. When the queue manager was attempting to generate the report message, it was unable to put it on the appropriate queue, and was also unable to put it on the dead-letter queue, so the report message could not be generated.
5. A failure of the queue manager occurred between the action being reported (arrival, delivery or expiry), and generation of the corresponding report message. (This does not happen for COD report messages if the application retrieves the original message within a unit of work, as the COD report message is generated within the same unit of work.)

Exception report messages may be held up in the same way for reasons 1, 2, and 3 above. However, when a message channel agent is unable to generate an exception report message (the report message cannot be put either on the reply queue or the dead-letter queue), the original message remains on the transmission queue at the sender, and the channel is closed. This occurs irrespective of whether the report message was to be generated at the sending or the receiving end of the channel.

If the original message is temporarily blocked (resulting in an exception report message being generated and the original message being put on a dead-letter queue), but the blockage clears and an application then reads the original message from the dead-letter queue and puts it again to its destination, the following may occur:

- Even though an exception report message has been generated, the original message eventually arrives successfully at its destination.
- More than one exception report message is generated in respect of a single original message, since the original message may encounter another blockage later.

Report messages for message segments: Report messages can be requested for messages that have segmentation allowed (see the description of the MQMF_SEGMENTATION_ALLOWED flag). If the queue manager finds it necessary to segment the message, a report message can be generated for each of the segments that subsequently encounters the relevant condition. Applications should therefore be prepared to receive multiple report messages for each type of report message requested. The *GroupId* field in the report message can be used to correlate the multiple reports with the group identifier of the original message, and the *Feedback* field used to identify the type of each report message.

If MQGMO_LOGICAL_ORDER is used to retrieve report messages for segments, be aware that reports of *different types* may be returned by the successive MQGET calls. For example, if both COA and COD reports are requested for a message that is segmented by the queue manager, the MQGET calls for the report messages may return the COA and COD report messages interleaved in an unpredictable fashion. This can be avoided by using the MQGMO_COMPLETE_MSG option (optionally with MQGMO_ACCEPT_TRUNCATED_MSG). MQGMO_COMPLETE_MSG causes the queue manager to reassemble report messages that have the same report type. For example, the first MQGET call might reassemble all of the COA messages relating to the original message, and the second MQGET call might reassemble all of the COD messages. Which is reassembled first depends on which type of report message happens to occur first on the queue.

Applications that themselves put segments can specify different report options for each segment. However, the following points should be noted:

- If the segments are retrieved using the MQGMO_COMPLETE_MSG option, only the report options in the *first* segment are honored by the queue manager.
- If the segments are retrieved one by one, and most of them have one of the MQRO_COD_★ options, but at least one segment does not, it will not be possible to use the MQGMO_COMPLETE_MSG option to retrieve the report messages with a single MQGET call, or use the MQGMO_ALL_SEGMENTS_AVAILABLE option to detect when all of the report messages have arrived.

In an MQ network, it is possible for the queue managers to have differing capabilities. If a report message for a segment is generated by a queue manager or MCA that does not support segmentation, the queue manager or MCA will not by default include the necessary segment information in the report message, and this may make it difficult to identify the original message that caused the report to be generated. This difficulty can be avoided by requesting data with the report message, that is, by specifying the appropriate MQRO_★_WITH_DATA or MQRO_★_WITH_FULL_DATA options. However, be aware that if MQRO_★_WITH_DATA is specified, *less than* 100 bytes of application message data may be returned to the application which retrieves the report message, if the report message is generated by a queue manager or MCA that does not support segmentation.

Contents of the message descriptor for a report message: When the queue manager or message channel agent generates a report message, it sets the fields in the message descriptor to the following values, and then puts the message in the normal way:

Field in MQMD	Value used
StrucId	MQMD_STRUC_ID
Version	MQMD_VERSION_1
Report	MQRO_NONE
MsgType	MQMT_REPORT
Expiry	MQEI_UNLIMITED

Feedback	As appropriate for the nature of the report (MQFB_COA, MQFB_COD, MQFB_EXPIRATION, or an MQRC_★ value)
Encoding	Copied from the original message descriptor
CodedCharSetId	Copied from the original message descriptor
Format	Copied from the original message descriptor
Priority	Copied from the original message descriptor
Persistence	Copied from the original message descriptor
MsgId	As specified by the report options in the original message descriptor
CorrelId	As specified by the report options in the original message descriptor
BackoutCount	0
ReplyToQ	Blanks
ReplyToQMgr	Name of queue manager
UserIdentifier	As set by the MQPMO_PASS_IDENTITY_CONTEXT option
AccountingToken	As set by the MQPMO_PASS_IDENTITY_CONTEXT option
ApplIdentityData	As set by the MQPMO_PASS_IDENTITY_CONTEXT option
PutApplType	MQAT_QMGR, or as appropriate for the message channel agent
PutApplName	First 28 bytes of the queue-manager name or message channel agent name. For report messages generated by the IMS bridge, this field contains the XCF group name and XCF member name of the IMS system to which the message relates.
PutDate	Date when report message is sent
PutTime	Time when report message is sent
ApplOriginData	Blanks
GroupId	Copied from the original message descriptor
MsgSeqNumber	Copied from the original message descriptor
Offset	Copied from the original message descriptor
MsgFlags	Copied from the original message descriptor
OriginalLength	Copied from the original message descriptor if not MQOL_UNDEFINED, and set to the length of the original message data otherwise

An application generating a report is recommended to set similar values, except for the following:

- The *ReplyToQMgr* field can be set to blanks (the queue manager will change this to the name of the local queue manager when the message is put).
- The context fields should be set using the option that would have been used for a reply, normally MQPMO_PASS_IDENTITY_CONTEXT.

Analyzing the report field: The *Report* field contains subfields; because of this, applications that need to check whether the sender of the message requested a particular report should use one of the techniques described in “Analyzing the report field” on page 491.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQRO_NONE.

MsgType (MQLONG)
Message type.

This indicates the type of the message. Message types are grouped as follows:

MQMT_SYSTEM_FIRST
Lowest value for system-defined message types.

MQMT_SYSTEM_LAST
Highest value for system-defined message types.

The following values are currently defined within the system range:

MQMT_DATAGRAM
Message not requiring a reply.

The message is one that does not require a reply.

MQMT_REQUEST
Message requiring a reply.

The message is one that requires a reply.

The name of the queue to which the reply should be sent must be specified in the *ReplyToQ* field. The *Report* field indicates how the *MsgId* and *CorrelId* of the reply are to be set.

MQMT_REPLY
Reply to an earlier request message.

The 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. The *Report* field of the request should be used to control how the *MsgId* and *CorrelId* of the reply are set.

Note: The queue manager does not enforce the request-reply relationship; this is an application responsibility.

MQMT_REPORT
Report message.

The message is reporting on some expected or unexpected occurrence, usually related to some other message (for example, a request message was received which contained data that was not valid). The message should be sent to the queue indicated by the *ReplyToQ* field of the message descriptor of the original message. The *Feedback* field should be set to indicate the nature of the report. The *Report* field of the original message can be used to control how the *MsgId* and *CorrelId* of the report message should be set.

Report messages generated by the queue manager or message channel agent are always sent to the *ReplyToQ* queue, with the *Feedback* and *CorrelId* fields set as described above.

Other values within the system range may be defined in future versions of the MQI, and are accepted by the MQPUT and MQPUT1 calls without error.

Application-defined values can also be used. They must be within the following range:

MQMT_APPL_FIRST
: Lowest value for application-defined message types.

MQMT_APPL_LAST
: Highest value for application-defined message types.

For the MQPUT and MQPUT1 calls, the *MsgType* value must be within either the system-defined range or the application-defined range; if it is not, the call fails with reason code MQRC_MSG_TYPE_ERROR.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls. The initial value of this field is MQMT_DATAGRAM.

Expiry (MQLONG)
: Message lifetime.

This is a period of time expressed in tenths of a second, set by the application that puts the message. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

The value is decremented to reflect the time the message spends on the destination queue, and also on any intermediate transmission queues if the put is to a remote queue. It may also be decremented by message channel agents to reflect transmission times, if these are significant. Likewise, an application forwarding this message to another queue might decrement the value if necessary, if it has retained the message for a significant time. However, the expiration time is treated as approximate, and the value need not be decremented to reflect small time intervals.

When the message is retrieved by an application using the MQGET call, the *Expiry* field represents the amount of the original expiry time that still remains.

After a message's expiry time has elapsed, it becomes eligible to be discarded by the queue manager. It is not defined, however, precisely when a message that is eligible for discarding is actually discarded. The discard often happens when a nonbrowse MQGET call occurs that would have returned the message had it not already expired (for example, a nonbrowse MQGET call with the *MatchOptions* field in MQGMO set to MQMO_NONE), but the discard can occur earlier or later than this.

A message that has expired is never returned to an application (either by a browse or a nonbrowse MQGET call), so the value in the *Expiry* field of the message descriptor after a successful MQGET call is either greater than zero, or the special value MQEI_UNLIMITED.

If a message is put on a remote queue, the message may expire (and be discarded) whilst it is on an intermediate transmission queue, before the message reaches the destination queue.

A report is generated when an expired message is discarded, if the message specified one of the MQRO_EXPIRATION_* report options. If none of these options is specified, no such report is generated; the message is assumed to be no longer relevant after this time period (perhaps because a later message has superseded it).

Any other program that discards messages based on expiry time must also send an appropriate report message if one was requested.

Notes:

1. If a message is put with an *Expiry* time of zero, the MQPUT or MQPUT1 call fails with reason code MQRC_EXPIRY_ERROR; no report message is generated in this case.
2. Since a message whose expiry time has elapsed may not actually be discarded until later, there may be messages on a queue that have passed their expiry time, and which are not therefore eligible for retrieval. These messages nevertheless count towards the number of messages on the queue for all purposes, including depth triggering.
3. An expiration report is generated, if requested, when the message is actually discarded, not when it becomes eligible for discarding.
4. Discarding of an expired message, and the generation of an expiration report if requested, are never part of the application's unit of work, even if the message was scheduled for discarding as a result of an MQGET call operating within a unit of work.
5. If a nearly-expired message is retrieved by an MQGET call within a unit of work, and the unit of work is subsequently backed out, the message may become eligible to be discarded before it can be retrieved again.
6. If a nearly-expired message is locked by an MQGET call with MQGMO_LOCK, the message may become eligible to be discarded before it can be retrieved by an MQGET call with MQGMO_MSG_UNDER_CURSOR; reason code MQRC_NO_MSG_UNDER_CURSOR is returned on this subsequent MQGET call if that happens.
7. Servers should not normally reflect the unused expiry time of a request in the reply; the default action should be to put the reply with MQEI_UNLIMITED. However, the default action for putting messages to a dead-letter (undelivered-message) queue is to preserve the outstanding expiry time of the message, and to continue to decrement it.
8. Trigger messages are always generated with MQEI_UNLIMITED.
9. A message (normally on a transmission queue) which has a *Format* name of MQFMT_XMIT_Q_HEADER has a second message descriptor within the MQXQH. It therefore has two *Expiry* fields associated with it. The following additional points should be noted in this case:
   - When an application puts a message on a remote queue, the queue manager places the message initially on a local transmission queue, and prefixes the application message data with an MQXQH structure. The queue manager sets the values of

the two *Expiry* fields to be the same as that specified by the application.

If an application puts a message directly on a local transmission queue, the message data must already begin with an MQXQH structure, and the format name must be MQFMT_XMIT_Q_HEADER (but the queue manager does not enforce this). In this case the application need not set the values of these two *Expiry* fields to be the same. (The queue manager does not check that the *Expiry* field within the MQXQH contains a valid value, or even that the message data is long enough to include it.)

- When a message with a *Format* name of MQFMT_XMIT_Q_HEADER is retrieved from a queue (whether this is a normal or a transmission queue), the queue manager decrements *both* these *Expiry* fields with the time spent waiting on the queue. No error is raised if the message data is not long enough to include the *Expiry* field in the MQXQH.
- The queue manager uses the *Expiry* field in the separate message descriptor (that is, not the one in the message descriptor embedded within the MQXQH structure) to test whether the message is eligible for discarding.
- If the initial values of the two *Expiry* fields were different, it is therefore possible for the *Expiry* time in the separate message descriptor when the message is retrieved to be greater than zero (so the message is not eligible for discarding), while the time according to the *Expiry* field in the MQXQH has elapsed. In this case the *Expiry* field in the MQXQH is set to zero.

The following special value is recognized:

MQEI_UNLIMITED
Unlimited lifetime.

The message has an unlimited expiration time.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQEI_UNLIMITED.

Feedback (MQLONG)
Feedback or reason 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.

The range of system-generated feedback codes

MQFB_SYSTEM_FIRST through MQFB_SYSTEM_LAST includes the special feedback codes (MQFB_*) listed below, and also the reason codes (MQRC_*) that can occur when the message cannot be put on the destination queue.

MQFB_APPL_FIRST
Lowest value for application-generated feedback.

MQFB_APPL_LAST
Highest value for application-generated feedback.

Applications that generate report messages should not use feedback codes in the system range (other than MQFB_QUIT), unless they wish to simulate report messages generated by the queue manager or message channel agent.

On the MQPUT or MQPUT1 calls, the value specified must be within either the system range or the application range. This is checked whatever the value of *MsgType*.

Special feedback codes are:

MQFB_EXPIRATION
Message expired.

Message was discarded because it had not been removed from the destination queue before its expiry time had elapsed.

MQFB_COA
Confirmation of arrival on the destination queue (see MQRO_COA).

MQFB_COD
Confirmation of delivery to the receiving application (see MQRO_COD).

MQFB_PAN
Positive action notification (see MQRO_PAN).

MQFB_NAN
Negative action notification (see MQRO_NAN).

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.

The following feedback codes are related to the IMS bridge:

MQFB_DATA_LENGTH_ZERO
Data length zero.

A segment length was zero in the application data of the message.

MQFB_DATA_LENGTH_NEGATIVE
Data length negative.

A segment length was negative in the application data of the message.

MQFB_DATA_LENGTH_TOO_BIG
Data length too big.

A segment length was too big in the application data of the message.

MQFB_BUFFER_OVERFLOW
Buffer overflow.

The value of one of the length fields would cause the data to overflow the MQSeries message buffer.

MQFB_LENGTH_OFF_BY_ONE
Length in error by one.

The value of one of the length fields was one byte too short.

MQFB_IIH_ERROR
MQIIH structure not valid or missing.

The *Format* field in MQMD specifies MQFMT_IMS, but the message does not begin with a valid MQIIH structure.

MQFB_NOT_AUTHORIZED_FOR_IMS
Userid not authorized for use in IMS.

The user ID contained in the message descriptor MQMD, or the password contained in the *Authenticator* field in the MQIIH structure, failed the validation performed by the IMS bridge. As a result the message was not passed to IMS.

MQFB_IMS_ERROR
Unexpected error returned by IMS.

An unexpected error was returned by IMS. Consult the MQSeries error log on the system on which the IMS bridge resides for more information about the error.

MQFB_IMS_FIRST
Lowest value for IMS-generated feedback.

IMS-generated feedback codes occupy the range MQFB_IMS_FIRST through MQFB_IMS_LAST. The IMS error code itself is *Feedback* minus MQFB_IMS_ERROR.

MQFB_IMS_LAST
Highest value for IMS-generated feedback.

For exception report messages, *Feedback* contains a reason code. Among possible reason codes are:

MQRC_PUT_INHIBITED
(2051, X'803') Put calls inhibited for the queue.

MQRC_Q_FULL
(2053, X'805') Queue already contains maximum number of messages.

MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.

MQRC_Q_SPACE_NOT_AVAILABLE
(2056, X'808') No space available on disk for queue.

MQRC_PERSISTENT_NOT_ALLOWED
(2048, X'800') Message on a temporary dynamic queue cannot be persistent.

MQRC_MSG_TOO_BIG_FOR_Q_MGR
(2031, X'7EF') Message length greater than maximum for queue manager.

MQRC_MSG_TOO_BIG_FOR_Q
(2030, X'7EE') Message length greater than maximum for queue.

For a full list of reason codes, see “Reason code” on page 383.

This is an output field for the MQGET call, and an input field for MQPUT and MQPUT1 calls. The initial value of this field is MQFB_NONE.

Encoding (MQLONG)
Data encoding.

This identifies the representation used for 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 default for the programming language and machine on which the application is running.

Note: The value of this constant is programming-language and environment specific.

The queue manager does not validate the contents of this field.

Applications that put messages should normally specify MQENC_NATIVE. Applications that retrieve messages should compare this field against the value MQENC_NATIVE; if the values differ, the application may need to convert numeric data in the message. See Appendix B, “Machine encodings” on page 485 for details of how this field is constructed.

If the MQGMO_CONVERT option is specified on the MQGET call, this field is an input/output field. The value specified by the application is the encoding to which the message data should be converted if necessary. If conversion is successful or unnecessary, the value is unchanged. If conversion is unsuccessful, the value after the MQGET call represents the encoding of the unconverted message that is returned to the application.

Otherwise, this is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQENC_NATIVE.

CodedCharSetId (MQLONG)
Coded character set identifier.

This specifies the coded character set identifier of character data in the application message data.

Note that character data in the message descriptor and the other MQI data structures must be in the character set used by the queue manager. This is defined by the queue manager's *CodedCharSetId* attribute; see “Attributes for the queue manager” on page 370 for details of this attribute.

The following values are defined:

MQCCSI_Q_MGR
: Queue manager's coded character set identifier.

  Character data in the application message data is in the queue manager's character set.

MQCCSI_EMBEDDED
: Embedded coded character set identifiers.

  The coded character-set identifier for character data in the message is embedded within the application message data itself. There can be any number of character-set identifiers embedded within the message, applying to different parts of the message.

  Specify this value only on the MQPUT and MQPUT1 calls. If it is specified on the MQGET call, it prevents conversion of the message.

On the MQPUT and MQPUT1 calls, the queue manager changes the value MQCCSI_Q_MGR to the value of the queue manager's *CodedCharSetId* attribute; as a result, the value MQCCSI_Q_MGR is never returned by the MQGET call. No other check is carried out on the value specified.

Applications that retrieve messages should compare this field against the value the application is expecting; if the values differ, the application may need to convert character data in the message.

If the MQGMO_CONVERT option is specified on the MQGET call, this field is an input/output field. The value specified by the application is the coded character-set identifier to which the message data should be converted if necessary. If conversion is successful or unnecessary, the value is unchanged (except that the value MQCCSI_Q_MGR is converted to the actual value). If conversion is unsuccessful, the value after the MQGET call represents the coded character-set identifier of the unconverted message that is returned to the application.

Otherwise, this is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQCCSI_Q_MGR.

Format (MQCHAR8)
: Format name.

  This is a 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 0 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.

  The name should be padded with blanks to the length of the field, or a null character used to terminate the name before the end of the field; the null and any subsequent characters are treated as blanks. Do not specify a name with leading or embedded blanks. For the MQGET call, the queue manager returns the name padded with blanks to the length of the field.

The queue manager does not check that the name complies with the recommendations described above.

Names beginning "MQ" have meanings that are defined by the queue manager; you should not use names beginning with these letters for your own formats. The queue manager built-in formats are:

MQFMT_NONE
: No format name.

  The nature of the application message data is undefined. This means that the data cannot be converted when the message is retrieved from a queue.

  Note: If MQGMO_CONVERT is specified on the MQGET call for a message that has a format name of MQFMT_NONE, and the character set or encoding of the message differs from that specified in the *MsgDesc* parameter, the message is still returned in the *Buffer* parameter (assuming no other errors), but the call completes with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

  For the C programming language, the constant MQFMT_NONE_ARRAY is also defined; this has the same value as MQFMT_NONE, but is an array of characters instead of a string.

MQFMT_ADMIN
: Command server request/reply message.

  The message is a command-server request or reply message in programmable command format (PCF). Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call. Refer to the *MQSeries Programmable Command Formats* for more information about programmable command format.

  For the C programming language, the constant MQFMT_ADMIN_ARRAY is also defined; this has the same value as MQFMT_ADMIN, but is an array of characters instead of a string.

MQFMT_CICS
: CICS information header.

  The message data begins with the CICS information header MQCIH, which is followed by the application data. The format name of the application data is given by the *Format* field in the MQCIH structure.

  On MVS/ESA, the MQGMO_CONVERT option can be specified on the MQGET call to convert messages that have format MQFMT_CICS.

  For the C programming language, the constant MQFMT_CICS_ARRAY is also defined; this has the same value as MQFMT_CICS, but is an array of characters instead of a string.

MQFMT_COMMAND_1
: Type 1 command reply message.

  The message is an MQSC command-server reply message containing the object count, completion code, and reason code.

Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_COMMAND_1_ARRAY is also defined; this has the same value as MQFMT_COMMAND_1, but is an array of characters instead of a string.

MQFMT_COMMAND_2
Type 2 command reply message.

The message is an MQSC command-server reply message containing information about the object(s) requested. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_COMMAND_2_ARRAY is also defined; this has the same value as MQFMT_COMMAND_2, but is an array of characters instead of a string.

MQFMT_DEAD_LETTER_HEADER
Dead-letter header.

The message data begins with the dead-letter header MQDLH. The data from the original message immediately follows the MQDLH structure. The format name of the original message data is given by the *Format* field in the MQDLH structure; see "MQDLH – Dead-letter header" on page 45 for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

COA and COD reports are not generated for messages which have a *Format* of MQFMT_DEAD_LETTER_HEADER.

For the C programming language, the constant MQFMT_DEAD_LETTER_HEADER_ARRAY is also defined; this has the same value as MQFMT_DEAD_LETTER_HEADER, but is an array of characters instead of a string.

MQFMT_DIST_HEADER
Distribution-list header.

The message data begins with the distribution-list header MQDH; this includes the arrays of MQOR and MQPMR records. The distribution-list header may be followed by additional data. The format of the additional data (if any) is given by the *Format* field in the MQDH structure; see "MQDH – Distribution header" on page 39 for details of this structure. Messages with format MQFMT_DIST_HEADER can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

This format is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

For the C programming language, the constant MQFMT_DIST_HEADER_ARRAY is also defined; this has the same value as MQFMT_DIST_HEADER, but is an array of characters instead of a string.

MQFMT_EVENT
Event message.

The message is an MQ event message that reports an event that occurred. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call. Event messages have the same structure as programmable commands; refer to the *MQSeries Programmable Command Formats* for more information about this structure.

For the C programming language, the constant MQFMT_EVENT_ARRAY is also defined; this has the same value as MQFMT_EVENT, but is an array of characters instead of a string.

MQFMT_IMS
IMS information header.

The message data begins with the IMS information header MQIIH, which is followed by the application data. The format name of the application data is given by the *Format* field in the MQIIH structure.

In the following environments, the MQGMO_CONVERT option can be specified on the MQGET call to convert messages that have format MQFMT_IMS: AIX, DOS client, HP-UX, MVS/ESA, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

For the C programming language, the constant MQFMT_IMS_ARRAY is also defined; this has the same value as MQFMT_IMS, but is an array of characters instead of a string.

MQFMT_IMS_VAR_STRING
IMS variable string.

The message is an IMS variable string, which is a string of the form `llzzccc`, where:

`ll` is a 2-byte length field specifying the total length of the IMS variable string item. This length is equal to the length of `ll` (2 bytes), plus the length of `zz` (2 bytes), plus the length of the character string itself. `ll` is a 2-byte binary integer in the encoding specified by the *Encoding* field.

`zz` is a 2-byte field containing flags that are significant to IMS. `zz` is a byte string consisting of two MQBYTE fields, and is transmitted without change from sender to receiver (that is, `zz` is not subject to any conversion).

`ccc` is a variable-length character string containing `ll-4` characters. `ccc` is in the character set specified by the the *CodedCharSetId* field.

In the following environments, the MQGMO_CONVERT option can be specified on the MQGET call to convert messages that have format MQFMT_IMS: AIX, DOS client, HP-UX, MVS/ESA, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

For the C programming language, the constant MQFMT_IMS_VAR_STRING_ARRAY is also defined; this has the same value as MQFMT_IMS_VAR_STRING, but is an array of characters instead of a string.

MQFMT_MD_EXTENSION
Message-descriptor extension.

The message data begins with the message-descriptor extension MQMDE, and is optionally followed by other data (usually the application message data). The format name, character set, and encoding of the data which follows the MQMDE is given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQMDE. See "MQMDE – Message descriptor extension" on page 153 for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

This format is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

For the C programming language, the constant MQFMT_MD_EXTENSION_ARRAY is also defined; this has the same value as MQFMT_MD_EXTENSION, but is an array of characters instead of a string.

MQFMT_PCF
User-defined message in programmable command format (PCF).

The message is a user-defined message that conforms to the structure of a programmable command format (PCF) message. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call. Refer to the *MQSeries Programmable Command Formats* for more information about programmable command format.

For the C programming language, the constant MQFMT_PCF_ARRAY is also defined; this has the same value as MQFMT_PCF, but is an array of characters instead of a string.

MQFMT_REF_MSG_HEADER
Reference message header.

The message data begins with the reference message header MQRMH, and is optionally followed by other data. The format name, character set, and encoding of the data is given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQRMH. See "MQMDE – Message descriptor extension" on page 153 for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

This format is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

For the C programming language, the constant MQFMT_REF_MSG_HEADER_ARRAY is also defined; this has the same value as MQFMT_REF_MSG_HEADER, but is an array of characters instead of a string.

MQFMT_STRING
Message consisting entirely of characters.

The application message data can be either an SBCS string (single-byte character set), or a DBCS string (double-byte character

set). Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_STRING_ARRAY is also defined; this has the same value as MQFMT_STRING, but is an array of characters instead of a string.

MQFMT_TRIGGER
Trigger message.

The message is a trigger message, described by the MQTM structure; see "MQTM – Trigger message" on page 209 for details of this structure. Messages of this format can be converted if the MQGMO_CONVERT option is specified on the MQGET call.

For the C programming language, the constant MQFMT_TRIGGER_ARRAY is also defined; this has the same value as MQFMT_TRIGGER, but is an array of characters instead of a string.

MQFMT_XMIT_Q_HEADER
Transmission queue header.

The message data begins with the transmission queue header MQXQH. The data from the original message immediately follows the MQXQH structure. The format name of the original message data is given by the *Format* field in the MQMD structure which is part of the transmission queue header MQXQH. See "MQXQH – Transmission queue header" on page 227 for details of this structure.

COA and COD reports are not generated for messages which have a *Format* of MQFMT_XMIT_Q_HEADER.

For the C programming language, the constant MQFMT_XMIT_Q_HEADER_ARRAY is also defined; this has the same value as MQFMT_XMIT_Q_HEADER, but is an array of characters instead of a string.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by MQ_FORMAT_LENGTH. The initial value of this field is MQFMT_NONE.

Priority (MQLONG)
Message priority.

For the MQPUT and MQPUT1 calls, the value must be greater than or equal to zero; zero is the lowest priority.

On MVS/ESA, the value must be in the range zero through *MaxPriority* (see "Attributes for the queue manager" on page 370).

The following special value can also be used:

MQPRI_PRIORITY_AS_Q_DEF
Default priority for queue.

The priority for the message is taken from the *DefPriority* attribute for the destination queue, as defined at the local queue manager. The value of *DefPriority* is copied into the *Priority* field when the

message is put. If *DefPriority* is changed subsequently, messages that have already been put are not affected.

If there is more than one definition in the queue-name resolution path, the default priority is taken from the value of this attribute in the *first* definition in the path (even if this is a queue-manager alias).

When replying to a message, applications should normally use for the reply message the priority of the request message. In other situations, defaulting to the queue definition allows priority tuning to be carried out without changing the application.

If a message is put with a priority greater than the maximum supported by the local queue manager (this maximum is given by the *MaxPriority* queue-manager attribute), the message is accepted by the queue manager, but placed on the queue at the queue manager's maximum priority; the MQPUT or MQPUT1 call completes with MQCC_WARNING and reason code MQRC_PRIORITY_EXCEEDS_MAXIMUM. However, the *Priority* field retains the value specified by the application which put the message.

On MVS/ESA, the call fails with completion code MQCC_FAILED and reason code MQRC_PRIORITY_ERROR if the message is put with a priority greater than the maximum supported by the local queue manager.

The value returned by the MQGET call is always greater than or equal to zero; the value MQPRI_PRIORITY_AS_Q_DEF is never returned.

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The initial value of this field is MQPRI_PRIORITY_AS_Q_DEF.

Persistence (MQLONG)
Message persistence.

For the 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. Because temporary dynamic queues *do not* survive restarts of the queue manager, persistent messages cannot be put on temporary dynamic queues; persistent messages can however be put on permanent dynamic queues, and predefined queues.

Once a persistent message has been put (or the unit of work committed, if the put request is part of a unit of work), the message is available on auxiliary storage until such time as the message is removed from the queue (or the unit of work committed, if the get request is part of a unit of work).

When a persistent message is sent to a remote queue, a store-and-forward mechanism is used to hold the message at each queue manager along the route to the destination, until the message is known to have arrived at the next queue manager.

MQPER_NOT_PERSISTENT
Message is not persistent.

The message does not survive restarts of the queue manager. This applies even if an intact copy of the message is found on auxiliary storage during the restart procedure.

MQPER_PERSISTENCE_AS_Q_DEF
Message has default persistence.

The persistence for the message is taken from the *DefPersistence* attribute for the destination queue, as defined at the local queue manager. The value of *DefPersistence* is copied into the *Persistence* field when the message is put. If *DefPersistence* is changed subsequently, messages that have already been put are not affected.

If there is more than one definition in the queue-name resolution path, the default persistence is taken from the value of this attribute in the *first* definition in the path (even if this is a queue-manager alias).

Both persistent and nonpersistent messages can exist on the same queue.

When replying to a message, applications should normally use for the reply message the persistence of the request message. In other situations, defaulting to the queue definition allows persistence to be changed without changing the application.

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 the MQPUT and MQPUT1 calls. The initial value of this field is MQPER_PERSISTENCE_AS_Q_DEF.

MsgId (MQBYTE24)
Message identifier.

This is a byte string that is used to distinguish one message from another. Generally, no two messages should have the same message identifier, although this is not disallowed by the queue manager. The message identifier is a permanent property of the message, and persists across restarts of the queue manager. Because the message identifier is a byte string and not a character string, the message identifier is *not* converted between character sets when the message flows from one queue manager to another.

For the MQPUT and MQPUT1 calls, if MQMI_NONE or MQPMO_NEW_MSG_ID is specified by the application, the queue manager generates a unique message identifier[4] when the message is put,

[4] A *MsgId* generated by the queue manager consists of a 4-byte product identifier ('AMQb' or 'CSQb' in either ASCII or EBCDIC, where 'b' represents a blank), followed by a product-specific implementation of a unique string. In MQSeries this contains the first 12 characters of the queue-manager name, and a value derived from the system clock. All queue managers that can intercommunicate must therefore have names that differ in the first 12 characters, in order to ensure that message identifiers are unique. The ability to generate a unique string also depends upon the system clock not being changed backward. To eliminate the possibility of a message identifier generated by the queue manager duplicating one generated by the application, the application should avoid generating identifiers with initial characters in the range A through I in ASCII or EBCDIC (X'41' through

and places it 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.

If the message is being put to a distribution list, the queue manager generates unique message identifiers as necessary, but the value of the *MsgId* field in MQMD is unchanged on return from the call, even if MQMI_NONE or MQPMO_NEW_MSG_ID was specified. If the application needs to know the message identifiers generated by the queue manager, the application must provide MQPMR records containing the *MsgId* field.

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. An application that is forwarding a message can use this facility 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, as described above
- Deliver the value to the application that issues the get request for the message
- Copy the value to the *CorrelId* field of any report message that it generates about this message (depending on the *Report* options)

When the queue manager or a message channel agent generates a report message, it sets the *MsgId* field in the way specified by the *Report* field of the original message, either MQRO_NEW_MSG_ID or MQRO_PASS_MSG_ID. Applications that generate report messages should also do this.

For the MQGET call, *MsgId* is one of the five fields that can be used to select a particular message to be retrieved from the queue. Normally the MQGET call returns the next message on the queue, but if a particular message is required, this can be obtained by specifying one or more of the five selection criteria, in any combination; these fields are:

MsgId
CorrelId
GroupId
MsgSeqNumber
Offset

The application sets one or more of these field to the values required, and then sets the corresponding MQMO_★ match options in the *MatchOptions* field in MQGMO to indicate that those fields should be used as selection criteria. Only messages that have the specified values in those fields are candidates for retrieval. The default for the *MatchOptions* field (if not altered by the application) is to match both the message identifier and the correlation identifier.

X'49' and X'C1' through X'C9'). However, the application is not prevented from generating identifiers with initial characters in these ranges.

Normally, the message returned is the *first* message on the queue that satisfies the selection criteria. But if MQGMO_BROWSE_NEXT is specified, the message returned is the *next* message that satisfies the selection criteria; the scan for this message starts with the message *following* the current cursor position.

Note: The queue is scanned sequentially for a message that satisfies the selection criteria, so retrieval times will be slower than if no selection criteria are specified, especially if many messages have to be scanned before a suitable one is found.

See Table 29 on page 75 for more information about how selection criteria are used in various situations.

Specifying MQMI_NONE as the message identifier has the same effect as *not* specifying MQMO_MATCH_MSG_ID, that is, *any* message identifier will match.

This field is ignored if the MQGMO_MSG_UNDER_CURSOR option is specified in the *GetMsgOpts* parameter on the MQGET call.

On return from an MQGET call, the *MsgId* field is set to the message identifier of the message returned (if any).

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 C programming language, the constant MQMI_NONE_ARRAY is also defined; this has the same value as MQMI_NONE, but is an array of characters instead of a string.

This is an input/output field for the MQGET, MQPUT, and MQPUT1 calls. The length of this field is given by MQ_MSG_ID_LENGTH. The initial value of this field is MQMI_NONE.

CorrelId (MQBYTE24)
Correlation identifier.

This is a byte string that the application can use to relate one message to another, or to relate the message to other work that the application is performing. The correlation identifier is a permanent property of the message, and persists across restarts of the queue manager. Because the correlation identifier is a byte string and not a character string, the correlation identifier is *not* converted between character sets when the message flows from one queue manager to another.

For the 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 issues the get request for the message.

If the application specifies MQPMO_NEW_CORREL_ID, the queue manager generates a unique correlation identifier which is sent with the message, and also returned to the sending application on output from the MQPUT or MQPUT1 call.

When the queue manager or a message channel agent generates a report message, it sets the *CorrelId* field in the way specified by the *Report* field

of the original message, either MQRO_COPY_MSG_ID_TO_CORREL_ID or MQRO_PASS_CORREL_ID. Applications which generate report messages should also do this.

For the MQGET call, *CorrelId* is one of the five fields that can be used to select a particular message to be retrieved from the queue. See the description of the *MsgId* field for details of how to specify values for this field.

Specifying MQCI_NONE as the correlation identifier has the same effect as *not* specifying MQMO_MATCH_CORREL_ID, that is, *any* correlation identifier will match.

If the MQGMO_MSG_UNDER_CURSOR option is specified in the *GetMsgOpts* parameter on the MQGET call, this field is ignored.

On return from an MQGET call, the *CorrelId* field is set to the correlation identifier of the message returned (if any).

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 C programming language, the constant MQCI_NONE_ARRAY is also defined; this has the same value as MQCI_NONE, but is an array of characters instead of a string.

MQCI_NEW_SESSION
Message is the start of a new session.

This value is recognized by the CICS bridge as indicating the start of a new session, that is, the start of a new sequence of messages.

For the C programming language, the constant MQCI_NEW_SESSION_ARRAY is also defined; this has the same value as MQCI_NEW_SESSION, but is an array of characters instead of a string.

For the MQGET call, this is an input/output field. For the MQPUT and MQPUT1 calls, this is an input field if MQPMO_NEW_CORREL_ID is *not* specified, and an output field if MQPMO_NEW_CORREL_ID *is* specified. The length of this field is given by MQ_CORREL_ID_LENGTH. The initial value of this field is MQCI_NONE.

BackoutCount (MQLONG)
Backout counter.

This is a count of the number of times the message has been previously returned by the MQGET call as part of a unit of work, and subsequently backed out. It is provided as an aid to the application in detecting processing errors that are based on message content. The count excludes MQGET calls that specified the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT options.

The accuracy of this count is affected by the *HardenGetBackout* local queue attribute; see “Attributes for local queues and model queues” on page 348.

On MVS/ESA, a value of 255 means that the message has been backed out 255 or more times; the value returned is never greater than 255.

On Tandem NSK, a backout count is maintained for each message. This count is an estimate of the number of times, within a single queue manager association, a message has been returned to an application on consecutive nonbrowse MQGET calls, and subsequently backed out under TMF control. The backout count is not saved to disk, and is therefore not guaranteed to be accurate. The backout count is reset to zero when an implicit or explicit MQDISC call is issued, when an MQGET call returns a different message, and at queue manager restart.

This is an output field for the MQGET call. It is ignored for the MQPUT and MQPUT1 calls. The initial value of this field is `0`.

ReplyToQ (MQCHAR48)
Name of reply queue.

This is 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*. This queue should not be a model queue, although the sending queue manager does not verify this when the message is put.

For the MQPUT and MQPUT1 calls, this field must not be blank if the *MsgType* field has the value MQMT_REQUEST, or if any reports are requested by the *Report* field. However, the value specified (or substituted; see below) is passed on to the application that issues the get request for the message, whatever the message type.

If the *ReplyToQMgr* field is blank, the local queue manager looks up the *ReplyToQ* name in its own queue definitions. If a local definition of a remote queue exists with this name, the *ReplyToQ* value in the transmitted message is replaced by the value of the *RemoteQName* attribute from the definition of the remote queue, and this value will be returned in the message descriptor when the receiving application issues an MQGET call for the message. If a local definition of a remote queue does not exist, *ReplyToQ* is unchanged.

If the name is specified, it may contain trailing blanks; the first null character and characters following it are treated as blanks. Otherwise, however, no check is made that the name satisfies the naming rules for queues; this is also true for the name transmitted, if the *ReplyToQ* is replaced in the transmitted message. The only check made is that a name has been specified, if the circumstances require it.

If a reply-to queue is not required, it is recommended (although this is not checked) that the *ReplyToQ* field should be set to blanks, or (in the C programming language) to the null string, or to one or more blanks followed by a null character; the field should not be left uninitialized.

For the MQGET call, the queue manager always returns the name padded with blanks to the length of the field.

If a message that requires a report message cannot be delivered, and the report message also cannot be delivered to the queue specified, both the original message and the report message go to the dead-letter

(undelivered-message) queue (see the *DeadLetterQName* attribute described in “Attributes for the queue manager” on page 370).

This is an output field for the MQGET call, and an input field for the MQPUT and MQPUT1 calls. The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ReplyToQMgr (MQCHAR48)
Name of reply queue manager.

This is the name of the queue manager to which the reply message or report message should be sent. *ReplyToQ* is the local name of a queue that is defined on this queue manager.

If the *ReplyToQMgr* field is blank, the local queue manager looks up the *ReplyToQ* name in its queue definitions. If a local definition of a remote queue exists with this name, the *ReplyToQMgr* value in the transmitted message is replaced by the value of the *RemoteQMgrName* attribute from the definition of the remote queue, and this value will be returned in the message descriptor when the receiving application issues an MQGET call for the message. If a local definition of a remote queue does not exist, the *ReplyToQMgr* that is transmitted with the message is the name of the local queue manager.

If the name is specified, it may contain trailing blanks; the first null character and characters following it are treated as blanks. Otherwise, however, no check is made that the name satisfies the naming rules for queue managers, or that this name is known to the sending queue manager; this is also true for the name transmitted, if the *ReplyToQMgr* is replaced in the transmitted message. For more information about names, see the *MQSeries Application Programming Guide*.

If a reply-to queue is not required, it is recommended (although this is not checked) that the *ReplyToQMgr* field should be set to blanks, or (in the C programming language) to the null string, or to one or more blanks followed by a null character; the field should not be left uninitialized.

For the 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. The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

UserIdentifier (MQCHAR12)
User identifier.

This is part of the **identity context** of the message; it identifies the user that originated this message. This information can be used in the *AlternateUserId* field of the *ObjDesc* parameter when opening an object, so that the authorization check is performed for the *UserIdentifier* user instead of the application performing the open.

The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it uses the *AlternateUserId* from the *ObjDesc* parameter if MQOO_ALTERNATE_USER_AUTHORITY was specified on the

corresponding MQOPEN call (or if MQPMO_ALTERNATE_USER_AUTHORITY is specified with the MQPUT1 call). Otherwise, it uses a user identifier determined by the environment:

- On MVS/ESA, it uses:
  - For MVS (batch), the user ID from the JES JOB card or started task
  - For TSO, the user ID propagated to the job during job submission
  - For CICS, the user ID associated with the task
  - For IMS, the user ID depends on the type of application:
    - For:
      - Nonmessage BMP regions
      - Nonmessage IFP regions
      - Message BMP and message IFP regions that have *not* issued a successful GU call

      the queue manager uses the user ID from the region JES JOB card or the TSO user ID. If these are blank or null, it uses the name of the program specification block (PSB).
    - For:
      - Message BMP and message IFP regions that *have* issued a successful GU call
      - MPP regions

      the queue manager uses one of:
      - The signed-on user ID associated with the message
      - The logical terminal (LTERM) name
      - The user ID from the region JES JOB card
      - The TSO user ID
      - The PSB name
- On OS/2, it uses the string “OS/2”.
- On OS/400, it uses the name of the signed-on user profile.
- On OpenVMS, Tandem NSK, and UNIX systems, it uses:
  - The application’s logon name
  - The effective user ID of the process if no logon is available
  - The user ID associated with the transaction, if the application is a CICS transaction
- On 16-bit Windows, it uses the string “WINDOWS”.
- On 32-bit Windows and Windows NT, it uses the first 12 characters of the logged-on user name.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *UserIdentifier* that was transmitted with the message. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_USER_ID_LENGTH. The initial value of this field is the null string in C, and 12 blank characters in other programming languages.

AccountingToken (MQBYTE32)
Accounting token.

This is part of the **identity context** of the message; it allows an application to cause work done as a result of the message to be appropriately charged.

The queue manager treats this information as a string of bits and does not check its content. When the queue manager generates this information, it sets:

- The first byte of the field to the length of the accounting information present in the remainder of the field; this length is in the range zero through 31, and is stored in the first byte as a binary integer.
- The second and subsequent bytes, as indicated by the length field, to the accounting information appropriate to the environment.
  - On MVS/ESA the accounting information is set to:
    - For MVS batch, the accounting information from the JES JOB card or from a JES ACCT statement in the EXEC card (comma separators are changed to X'FF'). This information is truncated, if necessary, to 31 bytes.
    - For TSO, the user's account number.
    - For CICS, the LU 6.2 unit of work identifier (UEPUOWDS) (26 bytes).
    - For IMS, the 8-character PSB name concatenated with the 16-character IMS recovery token.
  - On OS/400, the accounting information is set to the accounting code for the job.
  - On OpenVMS, Tandem NSK, and UNIX systems, the accounting information is set to the numeric user ID, in ASCII characters.
  - On OS/2, DOS client, Windows client, and Windows NT, the accounting information is set to the ASCII character '1'.
- All remaining bytes are set to binary zero.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. If neither MQPMO_SET_IDENTITY_CONTEXT nor MQPMO_SET_ALL_CONTEXT is specified, this field is ignored on input and is an output-only field. For more information on message context, see the *MQSeries Application Programming Guide*.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *AccountingToken* that was transmitted with the message. If the message has no context, the field is entirely binary zero.

This is an output field for the MQGET call.

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, and not as a string of characters.

The queue manager does nothing with the information in this field. The application must interpret the information if it wants to use the information for accounting purposes.

The following special value may be used:

MQACT_NONE
No accounting token is specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQACT_NONE_ARRAY is also defined; this has the same value as MQACT_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_ACCOUNTING_TOKEN_LENGTH. The initial value of this field is MQACT_NONE.

ApplIdentityData (MQCHAR32)
Application data relating to identity.

This is part of the **identity context** of the message; it is information that is defined by the application suite, and can be used to provide additional information about the message or its originator.

The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it is entirely blank.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. If a null character is present, the null and any following characters are converted to blanks by the queue manager. If neither MQPMO_SET_IDENTITY_CONTEXT nor MQPMO_SET_ALL_CONTEXT is specified, this field is ignored on input and is an output-only field. For more information on message context, see the *MQSeries Application Programming Guide*.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *ApplIdentityData* that was transmitted with the message. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_APPL_IDENTITY_DATA_LENGTH. The initial value of this field is the null string in C, and 32 blank characters in other programming languages.

PutApplType (MQLONG)
Type of application that put the message.

This is part of the **origin context** of the message. For more information on message context, see the *MQSeries Application Programming Guide*.

It may have one of the following standard types. User-defined types can also be used but should be restricted to values in the range MQAT_USER_FIRST through MQAT_USER_LAST.

MQAT_AIX
: AIX application (same value as MQAT_UNIX).

MQAT_CICS
: CICS transaction.

MQAT_DOS
: DOS client application.

MQAT_IMS
: IMS application.

MQAT_IMS_BRIDGE
: IMS bridge.

MQAT_MVS
: MVS or TSO application.

MQAT_OS2
: OS/2 or Presentation Manager application.

MQAT_OS400
: OS/400 application.

MQAT_QMGR
: Queue-manager-generated message.

MQAT_UNIX
: UNIX application.

MQAT_WINDOWS
: Windows client or 16-bit Windows application.

MQAT_WINDOWS_NT
: Windows NT or 32-bit Windows application.

MQAT_XCF
: XCF.

MQAT_DEFAULT
: Default application type.

  This is the default application type for the platform on which the application is running.

  Note: The value of this constant is environment-specific. Because of this, the application must be compiled using the header, include, or COPY files that are appropriate to the platform on which the application will run.

MQAT_UNKNOWN
: Unknown application type.

  This value can be used to indicate that the application type is unknown, even though other context information is present.

MQAT_USER_FIRST
: Lowest value for user-defined application type.

MQAT_USER_LAST
: Highest value for user-defined application type.

The following special value can also occur:

MQAT_NO_CONTEXT
No context information present in message.

This value is set by the queue manager when a message is put with no context (that is, the MQPMO_NO_CONTEXT context option is specified).

When a message is retrieved, *PutApplType* can be tested for this value to decide whether the message has context (it is recommended that *PutApplType* is never set to MQAT_NO_CONTEXT, by an application using MQPMO_SET_ALL_CONTEXT, if any of the other context fields are nonblank).

When the queue manager generates this information as a result of an application put, the field is set to a value that is determined by the environment. Note that on OS/400, it is set to MQAT_OS400; the queue manager never uses MQAT_CICS on OS/400.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *PutApplType* that was transmitted with the message. If the message has no context, the field is set to MQAT_NO_CONTEXT.

This is an output field for the MQGET call. The initial value of this field is MQAT_NO_CONTEXT.

PutApplName (MQCHAR28)
Name of application that put the message.

This is part of the **origin context** of the message. The format of the name depends on the *PutApplType*. For more information on message context, see the *MQSeries Application Programming Guide*.

When this field is set by the queue manager, (that is, for all options except MQPMO_SET_ALL_CONTEXT), it is set to value which is determined by the environment:

- On MVS/ESA, the queue manager uses:
  - For MVS batch, the 8-character job name from the JES JOB card
  - For TSO, the 7-character TSO user ID
  - For CICS, the 8-character applid, followed by the 4-character tranid
  - For IMS, the 8-character IMS system ID, followed by the 8-character PSB name
  - For XCF, the 8-character XCF group name, followed by the 16-character XCF member name
  - For a message generated by a queue manager, the first 28 characters of the queue manager name
  - For distributed queuing without CICS, the 8-character jobname of the channel initiator followed by the 8-character name of the module putting to the dead-letter queue followed by an 8-character task identifier.

The name or names are each padded to the right with blanks, as is any space in the remainder of the field. Where there is more than one name, there is no separator between them.

- On OS/2, DOS client, Windows client, and Windows NT, the queue manager uses:
  - For a CICS application, the CICS transaction name
  - For a non-CICS application, the rightmost 28 characters of the fully-qualified name of the executable
- On OS/400, the queue manager uses the fully-qualified job name.
- On OpenVMS, Tandem NSK, and UNIX systems, the queue manager uses:
  - For a CICS application, the CICS transaction name
  - For a non-CICS application, the rightmost 28 characters of the fully-qualified name of the executable, if this is available to the queue manager, and blanks otherwise

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *PutApplName* that was transmitted with the message. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_PUT_APPL_NAME_LENGTH. The initial value of this field is the null string in C, and 28 blank characters in other programming languages.

PutDate (MQCHAR8)
Date when message was put.

This is part of the **origin context** of the message. For more information on message context, see the *MQSeries Application Programming Guide*.

The format used for the date when this field is generated by the queue manager is:

YYYYMMDD

where the characters represent:

YYYY	year (four numeric digits)
MM	month of year (01 through 12)
DD	day of month (01 through 31)

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

On OS/2, the queue manager uses the TZ environment variable to calculate GMT. For more information on setting this variable, refer to the *MQSeries System Administration*.

If the message was put as part of a unit of work, the date is that when the message was put, and not the date when the unit of work was committed.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. The contents of the field are not checked by the queue manager, except that any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *PutDate* that was transmitted with the message. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_PUT_DATE_LENGTH. The initial value of this field is the null string in C, and 8 blank characters in other programming languages.

PutTime (MQCHAR8)
Time when message was put.

This is part of the **origin context** of the message. For more information on message context, see the *MQSeries Application Programming Guide*.

The format used for the time when this field is generated by the queue manager is:

HHMMSSTH

where the characters represent (in order):

HH hours (00 through 23)
MM minutes (00 through 59)
SS seconds (00 through 59; see note below)
T tenths of a second (0 through 9)
H hundredths of a second (0 through 9)

Note: If the system clock is synchronized to a very accurate time standard, it is possible on rare occasions for 60 or 61 to be returned for the seconds in *PutTime*. This happens when leap seconds are inserted into the global time standard.

Greenwich Mean Time (GMT) is used for the *PutDate* and *PutTime* fields, subject to the system clock being set accurately to GMT.

On OS/2, the queue manager uses the TZ environment variable to calculate GMT. For more information on setting this variable, refer to the *MQSeries System Administration*.

If the message was put as part of a unit of work, the time is that when the message was put, and not the time when the unit of work was committed.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. The contents of the field are not checked by the queue manager, except that any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *PutTime* that was transmitted with the message. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_PUT_TIME_LENGTH. The initial value of this field is the null string in C, and 8 blank characters in other programming languages.

ApplOriginData (MQCHAR4)
Application data relating to origin.

This is part of the **origin context** of the message; it is information that is defined by the application suite that can be used to provide additional information about the origin of the message. For example, it could be set by suitably authorized applications to indicate whether the identity data is trusted. For more information on message context, see the *MQSeries Application Programming Guide*.

The queue manager treats this information as character data, but does not define the format of it. When the queue manager generates this information, it is entirely blank.

For the MQPUT and MQPUT1 calls, this is an input/output field if MQPMO_SET_ALL_CONTEXT is specified in the *PutMsgOpts* parameter. Any information following a null character within the field is discarded. The null character and any following characters are converted to blanks by the queue manager. If MQPMO_SET_ALL_CONTEXT is not specified, this field is ignored on input and is an output-only field.

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *ApplOriginData* that was transmitted with the message. If the message has no context, the field is entirely blank.

This is an output field for the MQGET call. The length of this field is given by MQ_APPL_ORIGIN_DATA_LENGTH. The initial value of this field is the null string in C, and 4 blank characters in other programming languages.

The remaining fields in this structure are not present if *Version* is less than MQMD_VERSION_2.

GroupId (MQBYTE24)
Group identifier.

This is a byte string that is used to identify the particular message group or logical message to which the physical message belongs. *GroupId* is also used if segmentation is allowed for the message. In all of these cases, *GroupId* has a non-null value, and one or more of the following flags is set in the *MsgFlags* field:

MQMF_MSG_IN_GROUP
MQMF_LAST_MSG_IN_GROUP
MQMF_SEGMENT
MQMF_LAST_SEGMENT
MQMF_SEGMENTATION_ALLOWED

If none of these flags is set, *GroupId* has the special null value MQGI_NONE.

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, MQPMO_LOGICAL_ORDER is specified.
- On the MQGET call, MQMO_MATCH_GROUP_ID is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application requires more control, or the call is MQPUT1, the application must ensure that *GroupId* is set to an appropriate value.

Message groups and segments can be processed correctly only if the group identifier is unique. For this reason, *applications should not generate their own group identifiers*; instead, applications should do one of the following:

- If MQPMO_LOGICAL_ORDER is specified, the queue manager automatically generates a unique group identifier for the first message in the group or segment of the logical message, and uses that group identifier for the remaining messages in the group or segments of the logical message, so the application does not need to take any special action. This is the recommended procedure.
- If MQPMO_LOGICAL_ORDER is *not* specified, the application should request the queue manager to generate the group identifier, by setting *GroupId* to MQGI_NONE on the first MQPUT or MQPUT1 call for a message in the group or segment of the logical message. The group identifier returned by the queue manager on output from that call should then be used for the remaining messages in the group or segments of the logical message. If a message group contains segmented messages, the same group identifier must be used for all segments and messages in the group.

  When MQPMO_LOGICAL_ORDER is not specified, messages in groups and segments of logical messages can be put in any order (for example, in reverse order), but the group identifier must be allocated by the *first* MQPUT or MQPUT1 call that is issued for any of those messages.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in Table 44 on page 179. On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message if the object opened is a single queue and not a distribution list, but leaves it unchanged if the object opened is a distribution list. In the latter case, if the application needs to know the group identifiers generated, the application must provide MQPMR records containing the *GroupId* field.

On input to the MQGET call, the queue manager uses the value detailed in Table 29 on page 75. On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The following special value is defined:

MQGI_NONE
: No group identifier specified.

  The value is binary zero for the length of the field. This is the value that is used for messages that are not in groups, not segments of logical messages, and for which segmentation is not allowed.

  For the C programming language, the constant MQGI_NONE_ARRAY is also defined; this has the same value as MQGI_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_GROUP_ID_LENGTH. The initial value of this field is MQGI_NONE. This field is not present if *Version* is less than MQMD_VERSION_2.

MsgSeqNumber (MQLONG)
Sequence number of logical message within group.

Sequence numbers start at 1, and increase by 1 for each new logical message in the group, up to a maximum of 999 999 999. A physical message which is not in a group has a sequence number of 1.

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, MQPMO_LOGICAL_ORDER is specified.
- On the MQGET call, MQMO_MATCH_MSG_SEQ_NUMBER is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application requires more control, or the call is MQPUT1, the application must ensure that *MsgSeqNumber* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in Table 44 on page 179. On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message.

On input to the MQGET call, the queue manager uses the value detailed in Table 29 on page 75. On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The initial value of this field is one. This field is not present if *Version* is less than MQMD_VERSION_2.

Offset (MQLONG)
Offset of data in physical message from start of logical message.

This is the offset in bytes of the data in the physical message from the start of the logical message of which the data forms part. This data is called a *segment*. The offset is in the range 0 through 999 999 999. A physical message which is not a segment of a logical message has an offset of zero.

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, MQPMO_LOGICAL_ORDER is specified.
- On the MQGET call, MQMO_MATCH_OFFSET is *not* specified.

These are the recommended ways of using these calls for messages that are not report messages. However, if the application does not comply with these conditions, or the call is MQPUT1, the application must ensure that *Offset* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in Table 44 on page 179. On output from the MQPUT and MQPUT1 calls, the queue manager sets this field to the value that was sent with the message.

For a report message reporting on a segment of a logical message, the *OriginalLength* field (provided it is not MQOL_UNDEFINED) is used to update the offset in the segment information retained by the queue manager.

On input to the MQGET call, the queue manager uses the value detailed in Table 29 on page 75. On output from the MQGET call, the queue manager sets this field to the value for the message retrieved.

The initial value of this field is zero. This field is not present if *Version* is less than MQMD_VERSION_2.

MsgFlags (MQLONG)
Message flags.

These are flags that specify attributes of the message, or control its processing. The flags are divided into the following categories:

- Segmentation flag
- Status flags

These are described in turn.

Segmentation flag: When a message is too big for a queue, an attempt to put the message on the queue usually fails. Segmentation is a technique whereby the queue manager or application splits the message into smaller pieces called segments, and places each segment on the queue as a separate physical message. The application which retrieves the message can either retrieve the segments one by one, or request the queue manager to reassemble the segments into a single message which is returned by the MQGET call. The latter is achieved by specifying the MQGMO_COMPLETE_MSG option on the MQGET call, and supplying a buffer that is big enough to accommodate the complete message. (See "MQGMO – Get-message options" on page 56 for details of the MQGMO_COMPLETE_MSG option.) Segmentation of a message can occur at the sending queue manager, at an intermediate queue manager, or at the destination queue manager.

You can specify one of the following to control the segmentation of a message:

MQMF_SEGMENTATION_INHIBITED
Segmentation inhibited.

This option prevents the message being broken into segments by the queue manager. If specified for a message that is already a segment, this option prevents the segment being broken into smaller segments.

The value of this flag in binary zero. This is the default.

MQMF_SEGMENTATION_ALLOWED
Segmentation allowed.

This option allows the message to be broken into segments by the queue manager. If specified for a message that is already a segment, this option allows the segment to be broken into smaller segments. MQMF_SEGMENTATION_ALLOWED can be set without either MQMF_SEGMENT or MQMF_LAST_SEGMENT being set.

Note: Care is needed when messages are put with MQMF_SEGMENTATION_ALLOWED but without MQPMO_LOGICAL_ORDER. If the message is:

- not a segment, and
- not in a group, and
- not being forwarded,

the application must remember to reset the *GroupId* field to MQGI_NONE prior to *each* MQPUT or MQPUT1 call, in order to cause a unique group identifier to be generated by the queue manager for each message. If this is not done, unrelated messages could inadvertently end up with the same group identifier, which might lead to incorrect processing subsequently. See the descriptions of the *GroupId* field and the MQPMO_LOGICAL_ORDER option for more information about when the *GroupId* field must be reset.

The queue manager splits messages into segments as necessary in order to ensure that the segments (plus any header data that may be required) fit on the queue. However, there is a lower limit for the size of a segment generated by the queue manager (see below), and only the last segment created from a message can be smaller than this limit. The lower limit for the size of an application-generated segment is one byte. Segments generated by the queue manager may be of unequal length. The queue-manager processes the message as follows:

- User-defined formats are split on boundaries which are multiples of 16 bytes. This means that the queue manager will not generate segments that are smaller than 16 bytes (other than the last segment).
- Built-in formats other than MQFMT_STRING are split at points appropriate to the nature of the data present. However, the queue manager never splits a message in the middle of an MQ header structure. This means that a segment containing a single MQ header structure cannot be split further by the queue manager, and as a result the minimum possible segment size for that message is greater than 16 bytes.

  The second or later segment generated by the queue manager will begin with one of the following:

  – An MQ header structure
  – The start of the application message data
  – Part-way through the application message data
- MQFMT_STRING is split without regard for the nature of the data present (SBCS, DBCS, or mixed SBCS/DBCS). When the string is DBCS or mixed SBCS/DBCS, this may result in segments which cannot be converted from one character set to another (see below). The queue manager never splits MQFMT_STRING messages into segments that are smaller than 16 bytes (other than the last segment).
- The *Format*, *CodedCharSetId*, and *Encoding* fields in the MQMD of each segment are set by the queue manager to describe correctly the data present at the *start* of the segment; the format

name will be either the name of a built-in format, or the name of a user-defined format.

- The *Report* field in the MQMD of segments with *Offset* greater than zero are modified as follows:
  - For each report type, if the report option is MQRO_*_WITH_DATA, but the segment cannot possibly contain any of the first 100 bytes of user data (that is, the data following any MQ header structures that may be present), the report option is changed to MQRO_*.

The queue manager follows the above rules, but otherwise splits messages as it thinks fit; no assumptions should be made about the way that the queue manager will choose to split a particular message.

For *persistent* messages, the queue manager can perform segmentation only within a unit of work:

- If the MQPUT or MQPUT1 call is operating within a user-defined unit of work, that unit of work is used. If the call fails part way through the segmentation process, the queue manager removes any segments that were placed on the queue as a result of the failing call. However, the failure does not prevent the unit of work being committed successfully.
- If the call is operating outside a user-defined unit of work, and there is no user-defined unit of work in existence, the queue manager creates a unit of work just for the duration of the call. If the call is successful, the queue manager commits the unit of work automatically (the application does not need to do this). If the call fails, the queue manager backs out the unit of work.
- If the call is operating outside a user-defined unit of work, but a user-defined unit of work *does* exist, the queue manager is unable to perform segmentation. If the message does not require segmentation, the call can still succeed. But if the message *does* require segmentation, the call fails with reason code MQRC_UOW_NOT_AVAILABLE.

For *nonpersistent* messages, the queue manager does not require a unit of work to be available in order to perform segmentation.

Special consideration must be given to data conversion of messages which may be segmented:

- If data conversion is performed only by the receiving application on the MQGET call, and the application specifies the MQGMO_COMPLETE_MSG option, the data-conversion exit will be passed the complete message for the exit to convert, and the fact that the message was segmented will not be apparent to the exit.
- If the receiving application retrieves one segment at a time, the data-conversion exit will be invoked to convert one segment at a time. The exit must therefore be capable of converting the data in a segment independently of the data in any of the other segments.

If the nature of the data in the message is such that arbitrary segmentation of the data on 16-byte boundaries may result in segments which cannot be converted by the exit, or the format is MQFMT_STRING and the character set is DBCS or mixed SBCS/DBCS, the sending application should itself create and put the segments, specifying MQMF_SEGMENTATION_INHIBITED to suppress further segmentation. In this way, the sending application can ensure that each segment contains sufficient information to allow the data-conversion exit to convert the segment successfully.

- If sender conversion is specified for a sending message channel agent (MCA), the MCA converts only messages which are not segments of logical messages; the MCA never attempts to convert messages which are segments.

This flag is an input flag on the MQPUT and MQPUT1 calls, and an output flag on the MQGET call. On the latter call, the queue manager also echoes the value of the flag to the *Segmentation* field in MQGMO.

The initial value of this flag is MQMF_SEGMENTATION_INHIBITED.

Status flags: These are flags that indicate whether the physical message belongs to a message group, is a segment of a logical message, both, or neither. One or more of the following can be specified on the MQPUT or MQPUT1 call, or returned by the MQGET call:

MQMF_MSG_IN_GROUP
: Message is a member of a group.

MQMF_LAST_MSG_IN_GROUP
: Message is the last logical message in a group.

  If this flag is set, the queue manager turns on MQMF_MSG_IN_GROUP in the copy of MQMD that is sent with the message, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call.

  It is valid for a group to consist of only one logical message. If this is the case, MQMF_LAST_MSG_IN_GROUP is set, but the *MsgSeqNumber* field has the value one.

MQMF_SEGMENT
: Message is a segment of a logical message.

  When MQMF_SEGMENT is specified without MQMF_LAST_SEGMENT, the length of the application message data in the segment (*excluding* the lengths of any MQ header structures that may be present) must be at least one. If the length is zero, the MQPUT or MQPUT1 call fails with reason code MQRC_SEGMENT_LENGTH_ZERO.

MQMF_LAST_SEGMENT
: Message is the last segment of a logical message.

  If this flag is set, the queue manager turns on MQMF_SEGMENT in the copy of MQMD that is sent with the message, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call.

It is valid for a logical message to consist of only one segment. If this is the case, MQMF_LAST_SEGMENT is set, but the *Offset* field has the value zero.

When MQMF_LAST_SEGMENT is specified, it is permissible for the length of the application message data in the segment (*excluding* the lengths of any header structures that may be present) to be zero.

The application must ensure that these flags are set correctly when putting messages. If MQPMO_LOGICAL_ORDER is specified, or was specified on the preceding MQPUT call for the queue handle, the settings of the flags must be consistent with the group and segment information retained by the queue manager for the queue handle. The following conditions apply to *successive* MQPUT calls for the queue handle when MQPMO_LOGICAL_ORDER is specified:

- If there is no current group or logical message, all of these flags (and combinations of them) are valid.
- Once MQMF_MSG_IN_GROUP has been specified, it must remain on until MQMF_LAST_MSG_IN_GROUP is specified. The call fails with reason code MQRC_INCOMPLETE_GROUP if this condition is not satisfied.
- Once MQMF_SEGMENT has been specified, it must remain on until MQMF_LAST_SEGMENT is specified. The call fails with reason code MQRC_INCOMPLETE_MSG if this condition is not satisfied.
- Once MQMF_SEGMENT has been specified without MQMF_MSG_IN_GROUP, MQMF_MSG_IN_GROUP must remain *off* until after MQMF_LAST_SEGMENT has been specified. The call fails with reason code MQRC_INCOMPLETE_MSG if this condition is not satisfied.

Table 44 on page 179 shows the valid combinations of the flags, and the values used for various fields.

These flags are input flags on the MQPUT and MQPUT1 calls, and output flags on the MQGET call. On the latter call, the queue manager also echoes the values of the flags to the *GroupStatus* and *SegmentStatus* fields in MQGMO.

Default flags: The following can be specified to indicate that the message has default attributes:

MQMF_NONE
No message flags (default message attributes).

This inhibits segmentation, and indicates that the message is not in a group and is not a segment of a logical message. MQMF_NONE is defined to aid program documentation. It is not intended that this flag be used with any other, but as its value is zero, such use cannot be detected.

The *MsgFlags* field is partitioned into subfields; for details see Appendix C, “Report options and message flags” on page 489.

The initial value of this field is MQMF_NONE. This field is not present if *Version* is less than MQMD_VERSION_2.

OriginalLength (MQLONG)
Length of original message.

This field is of relevance only for report messages; it specifies the length of the message to which the report relates. If the report message is reporting on a segment, *OriginalLength* is the length of the segment, and *not* the length of the logical message of which the segment forms part, nor the length of the data in the report message.

OriginalLength should be set by the program which generates the report, or which segments the original message, but if that program does not set the field, *OriginalLength* has the following special value:

MQOL_UNDEFINED
Original length of message not defined.

This is an input field on the MQPUT and MQPUT1 calls, but the value provided by the application is used only in particular circumstances:

- If the message being put is a segment but not a report message, the queue manager ignores the field and uses the length of the application message data instead.
- If the message being put is a report message reporting on a segment, the queue manager accepts the value specified. The value must be:
  - Greater than zero if the segment is not the last segment
  - Not less than zero if the segment is the last segment
  - Not less than the length of data present in the message

  If these conditions are not satisfied, the call fails with reason code MQRC_ORIGINAL_LENGTH_ERROR.
- In all other cases, the queue manager ignores the field and uses the value MQOL_UNDEFINED instead.

This is an output field on the MQGET call.

The initial value of this field is MQOL_UNDEFINED. This field is not present if *Version* is less than MQMD_VERSION_2.

Table 35 (Page 1 of 2). Initial values of fields in MQMD

Field name	Name of constant	Value of constant
StrucId	MQMD_STRUC_ID	'MDbb' (See note 1)
Version	MQMD_VERSION_1	1
Report	MQRO_NONE	0
MsgType	MQMT_DATAGRAM	8
Expiry	MQEI_UNLIMITED	-1
Feedback	MQFB_NONE	0
Encoding	MQENC_NATIVE	See note 2
CodedCharSetId	MQCCSI_Q_MGR	0
Format	MQFMT_NONE	'bbbbbbbb'
Priority	MQPRI_PRIORITY_AS_Q_DEF	-1
Persistence	MQPER_PERSISTENCE_AS_Q_DEF	2

Table 35 (Page 2 of 2). Initial values of fields in MQMD

Field name	Name of constant	Value of constant
MsgId	MQMI_NONE	Nulls
CorrelId	MQCI_NONE	Nulls
BackoutCount	None	0
ReplyToQ	None	Blanks (See note 3)
ReplyToQMgr	None	Blanks
UserIdentifier	None	Blanks
AccountingToken	MQACT_NONE	Nulls
ApplIdentityData	None	Blanks
PutApplType	MQAT_NO_CONTEXT	0
PutApplName	None	Blanks
PutDate	None	Blanks
PutTime	None	Blanks
ApplOriginData	None	Blanks
GroupId	MQGI_NONE	Nulls
MsgSeqNumber	None	1
Offset	None	0
MsgFlags	MQMF_NONE	0
OriginalLength	MQOL_UNDEFINED	-1

Notes:

1. The symbol 'b' represents a single blank character.
2. The value of this constant is environment-specific.
3. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
4. In the C programming language, the macro variable MQMD_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQMD MyMD = {MQMD_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQMD {
  MQCHAR4   StrucId;          /* Structure identifier */
  MQLONG    Version;          /* Structure version number */
  MQLONG    Report;           /* Report options */
  MQLONG    MsgType;          /* Message type */
  MQLONG    Expiry;           /* Expiry time */
  MQLONG    Feedback;         /* Feedback or reason 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;    /* User identifier */
  MQBYTE32  AccountingToken;   /* Accounting token */
  MQCHAR32  ApplIdentityData;  /* Application data relating to
                                  identity */
  MQLONG    PutApplType;       /* Type of application that put the
                                  message */
  MQCHAR28  PutApplName;       /* Name of application that put the
                                  message */
  MQCHAR8   PutDate;           /* Date when message was put */
  MQCHAR8   PutTime;           /* Time when message was put */
  MQCHAR4   ApplOriginData;    /* Application data relating to origin */
  MQBYTE24  GroupId;           /* Group identifier */
  MQLONG    MsgSeqNumber;      /* Sequence number of logical message
                                  within group */
  MQLONG    Offset;            /* Offset of data in physical message
                                  from start of logical message */
  MQLONG    MsgFlags;          /* Message flags */
  MQLONG    OriginalLength;    /* Length of original message */
 } MQMD;
```

COBOL language declaration

```
**   MQMD structure
  10 MQMD.
**    Structure identifier
   15 MQMD-STRUCID          PIC X(4).
**    Structure version number
   15 MQMD-VERSION          PIC S9(9) BINARY.
**    Report options
   15 MQMD-REPORT           PIC S9(9) BINARY.
**    Message type
   15 MQMD-MSGTYPE          PIC S9(9) BINARY.
**    Expiry time
   15 MQMD-EXPIRY           PIC S9(9) BINARY.
**    Feedback or reason code
   15 MQMD-FEEDBACK         PIC S9(9) BINARY.
**    Data encoding
   15 MQMD-ENCODING         PIC S9(9) BINARY.
**    Coded character set identifier
   15 MQMD-CODEDCHARSETID   PIC S9(9) BINARY.
**    Format name
   15 MQMD-FORMAT           PIC X(8).
**    Message priority
   15 MQMD-PRIORITY         PIC S9(9) BINARY.
**    Message persistence
   15 MQMD-PERSISTENCE      PIC S9(9) BINARY.
**    Message identifier
   15 MQMD-MSGID            PIC X(24).
**    Correlation identifier
   15 MQMD-CORRELID         PIC X(24).
**    Backout counter
   15 MQMD-BACKOUTCOUNT     PIC S9(9) BINARY.
**    Name of reply-to queue
   15 MQMD-REPLYTOQ         PIC X(48).
**    Name of reply queue manager
```

```
   15 MQMD-REPLYTOQMGR      PIC X(48).
**    User identifier
   15 MQMD-USERIDENTIFIER   PIC X(12).
**    Accounting token
   15 MQMD-ACCOUNTINGTOKEN  PIC X(32).
**    Application data relating to identity
   15 MQMD-APPLIDENTITYDATA PIC X(32).
**    Type of application that put the message
   15 MQMD-PUTAPPLTYPE      PIC S9(9) BINARY.
**    Name of application that put the message
   15 MQMD-PUTAPPLNAME      PIC X(28).
**    Date when message was put
   15 MQMD-PUTDATE          PIC X(8).
**    Time when message was put
   15 MQMD-PUTTIME          PIC X(8).
**    Application data relating to origin
   15 MQMD-APPLORIGINDATA   PIC X(4).
**    Group identifier
   15 MQMD-GROUPID          PIC X(24).
**    Sequence number of logical message within group
   15 MQMD-MSGSEQNUMBER     PIC S9(9) BINARY.
**    Offset of data in physical message from start of logical
**    message
   15 MQMD-OFFSET           PIC S9(9) BINARY.
**    Message flags
   15 MQMD-MSGFLAGS         PIC S9(9) BINARY.
**    Length of original message
   15 MQMD-ORIGINALLENGTH   PIC S9(9) BINARY.
```

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

```
dcl
 1 MQMD based,
  3 StrucId          char(4),       /* Structure identifier */
  3 Version          fixed bin(31), /* Structure version number */
  3 Report           fixed bin(31), /* Report options */
  3 MsgType          fixed bin(31), /* Message type */
  3 Expiry           fixed bin(31), /* Expiry time */
  3 Feedback         fixed bin(31), /* Feedback or reason code */
  3 Encoding         fixed bin(31), /* Data encoding */
  3 CodedCharSetId   fixed bin(31), /* Coded character set identifier */
  3 Format           char(8),       /* Format name */
  3 Priority         fixed bin(31), /* Message priority */
  3 Persistence      fixed bin(31), /* Message persistence */
  3 MsgId            char(24),      /* Message identifier */
  3 CorrelId         char(24),      /* Correlation identifier */
  3 BackoutCount     fixed bin(31), /* Backout counter */
  3 ReplyToQ         char(48),      /* Name of reply-to queue */
  3 ReplyToQMgr      char(48),      /* Name of reply queue manager */
  3 UserIdentifier   char(12),      /* User identifier */
  3 AccountingToken  char(32),      /* Accounting token */
  3 ApplIdentityData char(32),      /* Application data relating to
                                       identity */
  3 PutApplType      fixed bin(31), /* Type of application that put the
                                       message */
  3 PutApplName      char(28),      /* Name of application that put the
                                       message */
  3 PutDate          char(8),       /* Date when message was put */
```

```
  3 PutTime          char(8),       /* Time when message was put */
  3 ApplOriginData   char(4),       /* Application data relating to
                                       origin */
  3 GroupId          char(24),      /* Group identifier */
  3 MsgSeqNumber     fixed bin(31), /* Sequence number of logical
                                       message within group */
  3 Offset           fixed bin(31), /* Offset of data in physical
                                       message from start of logical
                                       message */
  3 MsgFlags         fixed bin(31), /* Message flags */
  3 OriginalLength   fixed bin(31); /* Length of original message */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQMD                        DSECT
MQMD_STRUCID                DS   CL4      Structure identifier
MQMD_VERSION                DS   F        Structure version number
MQMD_REPORT                 DS   F        Report options
MQMD_MSGTYPE                DS   F        Message type
MQMD_EXPIRY                 DS   F        Expiry time
MQMD_FEEDBACK               DS   F        Feedback or reason code
MQMD_ENCODING               DS   F        Data encoding
MQMD_CODEDCHARSETID         DS   F        Coded character set
*                                         identifier
MQMD_FORMAT                 DS   CL8      Format name
MQMD_PRIORITY               DS   F        Message priority
MQMD_PERSISTENCE            DS   F        Message persistence
MQMD_MSGID                  DS   XL24     Message identifier
MQMD_CORRELID               DS   XL24     Correlation identifier
MQMD_BACKOUTCOUNT           DS   F        Backout counter
MQMD_REPLYTOQ               DS   CL48     Name of reply-to queue
MQMD_REPLYTOQMGR            DS   CL48     Name of reply queue manager
MQMD_USERIDENTIFIER         DS   CL12     User identifier
MQMD_ACCOUNTINGTOKEN        DS   XL32     Accounting token
MQMD_APPLIDENTITYDATA       DS   CL32     Application data relating to
*                                         identity
MQMD_PUTAPPLTYPE            DS   F        Type of application that put
*                                         the message
MQMD_PUTAPPLNAME            DS   CL28     Name of application that put
*                                         the message
MQMD_PUTDATE                DS   CL8      Date when message was put
MQMD_PUTTIME                DS   CL8      Time when message was put
MQMD_APPLORIGINDATA         DS   CL4      Application data relating to
*                                         origin
MQMD_LENGTH                 EQU  *-MQMD   Length of structure
                            ORG  MQMD
MQMD_AREA                   DS   CL(MQMD_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQMD^DEF (*);
   BEGIN
STRUCT         STRUCID;
BEGIN STRING BYTE [0:3]; END;
INT(32)        VERSION;
INT(32)        REPORTOPTIONS;
INT(32)        MSGTYPE;
INT(32)        EXPIRY;
```

```
|                       INT(32)        FEEDBACK;
|                       INT(32)        ENCODING;
|                       INT(32)        CODEDCHARSETID;
|                       STRUCT         FORMAT;
|                       BEGIN STRING BYTE [0:7]; END;
|                       INT(32)        PRIORITY;
|                        INT(32)       PERSISTENCE;
|                       STRUCT         MSGID;
|                       BEGIN STRING BYTE [0:23]; END;
|                       STRUCT         CORRELID;
|                       BEGIN STRING BYTE [0:23]; END;
|                       INT(32)        BACKOUTCOUNT;
|                       STRUCT         REPLYTOQ;
|                       BEGIN STRING BYTE [0:47]; END;
|                       STRUCT         REPLYTOQMGR;
|                       BEGIN STRING BYTE [0:47]; END;
|                       STRUCT         USERIDENTIFIER;
|                       BEGIN STRING BYTE [0:11]; END;
|                       STRUCT         ACCOUNTINGTOKEN;
|                       BEGIN STRING BYTE [0:31]; END;
|                       STRUCT         APPLIDENTITYDATA;
|                       BEGIN STRING BYTE [0:31]; END;
|                       INT(32)        PUTAPPLTYPE;
|                       STRUCT         PUTAPPLNAME;
|                       BEGIN STRING BYTE [0:27]; END;
|                       STRUCT         PUTDATE;
|                       BEGIN STRING BYTE [0:7]; END;
|                       STRUCT         PUTTIME;
|                       BEGIN STRING BYTE [0:7]; END;
|                       STRUCT         APPLORIGINDATA;
|                       BEGIN STRING BYTE [0:3]; END;
|                          END;
```

MQMDE – Message descriptor extension

The following table summarizes the fields in the structure.

Table 36. Fields in MQMDE

Field	Description	Page
StrucId	Structure identifier	156
Version	Structure version number	156
StrucLength	Length of MQMDE structure	156
Encoding	Encoding of the data following the MQMDE	156
CodedCharSetId	Character-set identifier of the data following the MQMDE	156
Format	Format name of the data following the MQMDE	156
Flags	General flags	157
GroupId	Group identifier	157
MsgSeqNumber	Sequence number of logical message within group	157
Offset	Offset of data in physical message from start of logical message	157
MsgFlags	Message flags	157
OriginalLength	Length of original message	157

The MQMDE structure describes the data that sometimes occurs preceding the application message data. Normal applications should use a version-2 MQMD, in which case they will not encounter an MQMDE structure. However, specialized applications, and applications that continue to use a version-1 MQMD, may encounter an MQMDE in some situations.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

The MQMDE structure contains those MQMD fields that exist in the version-2 MQMD, but not in the version-1 MQMD. It can occur in the following circumstances:

- Specified on the MQPUT and MQPUT1 calls
- Returned by the MQGET call
- In messages on transmission queues

These are described below.

MQMDE specified on MQPUT and MQPUT1 calls: On the MQPUT and MQPUT1 calls, if the application provides a version-1 MQMD, the application can optionally prefix the message data with an MQMDE, setting the *Format* field in MQMD to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. If the application does not provide an MQMDE, the queue manager assumes default values for the fields in the MQMDE. The default values that the queue manager uses are the same as the initial values for the structure – see Table 38 on page 158.

If the application provides a version-2 MQMD *and* prefixes the application message data with an MQMDE, the structures are processed as shown in Table 37 on page 155.

There is one special case. If the application uses a version-2 MQMD to put a message that is a segment (that is, the MQMF_SEGMENT or MQMF_LAST_SEGMENT flag is set), and the format name in the MQMD is MQFMT_DEAD_LETTER_HEADER, the queue manager generates an MQMDE structure and inserts it *between* the MQDLH structure and the data that follows it. In the MQMD that the queue manager retains with the message, the version-2 fields are set to their default values.

The data in the MQMDE structure must be in the queue manager's character set and encoding. The former is given by the *CodedCharSetId* queue-manager attribute (see "Attributes for the queue manager" on page 370), while in most cases the latter is given by the value of MQENC_NATIVE. If this condition is not satisfied, the MQMDE is accepted but not honored, that is, the MQMDE is treated as message data.

Note: On OS/2 and Windows NT, applications compiled with Micro Focus COBOL use a value of MQENC_NATIVE that is different from the queue-manager's encoding. Although numeric fields in the MQMD structure on the MQPUT, MQPUT1, and MQGET calls must be in the Micro Focus COBOL encoding, numeric fields in the MQMDE structure must be in the queue-manager's encoding. This latter is given by MQENC_NATIVE for the C programming language, and has the value 546.

Several of the fields that exist in the version-2 MQMD but not the version-1 MQMD are input/output fields on MQPUT and MQPUT1. However, the queue manager *does not* return any values in the equivalent fields in the MQMDE on output from the MQPUT and MQPUT1 calls; if the application requires those output values, it must use a version-2 MQMD.

MQMDE returned by MQGET call: On the MQGET call, if the application provides a version-1 MQMD, the queue manager prefixes the message returned with an MQMDE, but only if one or more of the fields in the MQMDE has a nondefault value. The *Format* field in MQMD will have the value MQFMT_MD_EXTENSION to indicate that an MQMDE is present.

If the application provides an MQMDE at the start of the *Buffer* parameter, the MQMDE is ignored. On return from the MQGET call, it will have been replaced by the MQMDE for the message (if one is needed), or overwritten by the application message data (if the MQMDE is not needed).

If an MQMDE is returned by the MQGET call, the data in the MQMDE will usually be in the queue manager's character set and encoding. The one exception is if the MQMDE was treated as data on the MQPUT or MQPUT1 call (see Table 37 on page 155 for the circumstances that can cause this).

Note: On OS/2 and Windows NT, applications compiled with Micro Focus COBOL use a value of MQENC_NATIVE that is different from the queue-manager's encoding (see above).

MQMDE in messages on transmission queues: Messages on transmission queues are prefixed with the MQXQH structure, which contains within it a version-1

Table 37. Queue-manager action when MQMDE specified on MQPUT or MQPUT1. This table shows the action taken by the queue manager when the application specifies an MQMDE structure at the start of the application message data on the MQPUT or MQPUT1 call.

MQMD version	Values of version-2 fields	Values of corresponding fields in MQMDE	Action taken by queue manager
1	–	Valid	MQMDE is honored
1	–	Not valid	Call fails with an appropriate reason code
1	–	MQMDE is in the wrong character set or encoding, or is an unsupported version	MQMDE is treated as message data
2	Default	Valid	MQMDE is honored
2	Default	Not valid	Call fails with an appropriate reason code
2	Default	MQMDE is in the wrong character set or encoding, or is an unsupported version	MQMDE is treated as message data
2	Not default	Valid, and same as MQMD	MQMDE is honored
2	Not default	Valid, but different from MQMD	MQMDE is treated as message data
2	Not default	Not valid	Call fails with an appropriate reason code
2	Not default	MQMDE is in the wrong character set or encoding, or is an unsupported version	MQMDE is treated as message data

MQMD. An MQMDE may also be present, positioned between the MQXQH structure and application message data, but it will usually be present only if one or more of the fields in the MQMDE has a nondefault value.

Other MQ header structures can also occur between the MQXQH structure and the application message data. For example, when the dead-letter header MQDLH is present, and the message is not a segment, the order is:

- MQXQH (containing a version-1 MQMD)
- MQMDE
- MQDLH
- application message data

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQMDE_STRUC_ID
Identifier for message descriptor extension structure.

For the C programming language, the constant MQMDE_STRUC_ID_ARRAY is also defined; this has the same value as MQMDE_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQMDE_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQMDE_VERSION_2
Version-2 message descriptor extension structure.

The following constant specifies the version number of the current version:

MQMDE_CURRENT_VERSION
Current version of message descriptor extension structure.

The initial value of this field is MQMDE_VERSION_2.

StrucLength (MQLONG)
Length of MQMDE structure.

The following value is defined:

MQMDE_LENGTH_2
Length of version-2 message descriptor extension structure.

The initial value of this field is MQMDE_LENGTH_2.

Encoding (MQLONG)
Encoding of the data following the MQMDE.

The queue manager does not check the value of this field. See the *Encoding* field described in “MQMD – Message descriptor” on page 98 for more information about data encodings.

The initial value of this field is MQENC_NATIVE.

CodedCharSetId (MQLONG)
Character-set identifier of the data following the MQMDE.

The queue manager does not check the value of this field.

The initial value of this field is `0`.

Format (MQCHAR8)
Format name of the data following the MQMDE.

The queue manager does not check the value of this field. See the *Format* field described in “MQMD – Message descriptor” on page 98 for more information about format names.

The initial value of this field is MQFMT_NONE.

Flags (MQLONG)
General flags.

The following flag can be specified:

MQMDEF_NONE
No flags.

The initial value of this field is MQMDEF_NONE.

GroupId (MQBYTE24)
Group identifier.

See the *GroupId* field described in “MQMD – Message descriptor” on page 98. The initial value of this field is MQGI_NONE.

MsgSeqNumber (MQLONG)
Sequence number of logical message within group.

See the *MsgSeqNumber* field described in “MQMD – Message descriptor” on page 98. The initial value of this field is 1.

Offset (MQLONG)
Offset of data in physical message from start of logical message.

See the *Offset* field described in “MQMD – Message descriptor” on page 98. The initial value of this field is 0.

MsgFlags (MQLONG)
Message flags.

See the *MsgFlags* field described in “MQMD – Message descriptor” on page 98. The initial value of this field is MQMF_NONE.

OriginalLength (MQLONG)
Length of original message.

See the *OriginalLength* field described in “MQMD – Message descriptor” on page 98. The initial value of this field is MQOL_UNDEFINED.

Table 38. Initial values of fields in MQMDE

Field name	Name of constant	Value of constant
StrucId	MQMDE_STRUC_ID	'MDEb' (See note 1)
Version	MQMDE_VERSION_2	2
StrucLength	MQMDE_LENGTH_2	72
Encoding	MQENC_NATIVE	See note 2
CodedCharSetId	None	0
Format	MQFMT_NONE	'bbbbbbbb'
Flags	MQMDEF_NONE	0
GroupId	MQGI_NONE	Nulls
MsgSeqNumber	None	1
Offset	None	0
MsgFlags	MQMF_NONE	0
OriginalLength	MQOL_UNDEFINED	-1

Notes:

1. The symbol 'b' represents a single blank character.
2. The value of this constant is environment-specific.
3. In the C programming language, the macro variable MQMDE_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQMDE MyMDE = {MQMDE_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQMDE {
  MQCHAR4   StrucId;         /* Structure identifier */
  MQLONG    Version;         /* Structure version number */
  MQLONG    StrucLength;     /* Length of MQMDE structure */
  MQLONG    Encoding;        /* Encoding of message data */
  MQLONG    CodedCharSetId;  /* Coded character-set identifier of
                                message data */
  MQCHAR8   Format;          /* Format name of message data */
  MQLONG    Flags;           /* General flags */
  MQBYTE24  GroupId;         /* Group identifier */
  MQLONG    MsgSeqNumber;    /* Sequence number of logical message
                                within group */
  MQLONG    Offset;          /* Offset of data in physical message from
                                start of logical message */
  MQLONG    MsgFlags;        /* Message flags */
  MQLONG    OriginalLength;  /* Length of original message */
 } MQMDE;
```

COBOL language declaration

```
**   MQMDE structure
  10 MQMDE.
**    Structure identifier
   15 MQMDE-STRUCID        PIC X(4).
**    Structure version number
   15 MQMDE-VERSION        PIC S9(9) BINARY.
**    Length of MQMDE structure
   15 MQMDE-STRUCLENGTH    PIC S9(9) BINARY.
**    Encoding of message data
   15 MQMDE-ENCODING       PIC S9(9) BINARY.
**    Coded character-set identifier of message data
   15 MQMDE-CODEDCHARSETID PIC S9(9) BINARY.
**    Format name of message data
   15 MQMDE-FORMAT         PIC X(8).
**    General flags
   15 MQMDE-FLAGS          PIC S9(9) BINARY.
**    Group identifier
   15 MQMDE-GROUPID        PIC X(24).
**    Sequence number of logical message within group
   15 MQMDE-MSGSEQNUMBER   PIC S9(9) BINARY.
**    Offset of data in physical message from start of logical
**    message
   15 MQMDE-OFFSET         PIC S9(9) BINARY.
**    Message flags
   15 MQMDE-MSGFLAGS       PIC S9(9) BINARY.
**    Length of original message
   15 MQMDE-ORIGINALLENGTH PIC S9(9) BINARY.
```

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

```
dcl
 1 MQMDE based,
  3 StrucId        char(4),       /* Structure identifier */
  3 Version        fixed bin(31), /* Structure version number */
  3 StrucLength    fixed bin(31), /* Length of MQMDE structure */
  3 Encoding       fixed bin(31), /* Encoding of message data */
  3 CodedCharSetId fixed bin(31), /* Coded character-set identifier of
                                     message data */
  3 Format         char(8),       /* Format name of message data */
  3 Flags          fixed bin(31), /* General flags */
  3 GroupId        char(24),      /* Group identifier */
  3 MsgSeqNumber   fixed bin(31), /* Sequence number of logical message
                                     within group */
  3 Offset         fixed bin(31), /* Offset of data in physical message
                                     from start of logical message */
  3 MsgFlags       fixed bin(31), /* Message flags */
  3 OriginalLength fixed bin(31); /* Length of original message */
```

MQOD – Object descriptor

The following table summarizes the fields in the structure.

Table 39. Fields in MQOD

Field	Description	Page
StrucId	Structure identifier	161
Version	Structure version number	161
ObjectType	Object type	161
ObjectName	Object name	162
ObjectQMgrName	Object queue manager name	162
DynamicQName	Dynamic queue name	163
AlternateUserId	Alternate user identifier	163
Note: The remaining fields are supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.		
RecsPresent	Number of object records present	164
KnownDestCount	Number of local queues opened successfully	164
UnknownDestCount	Number of remote queues opened successfully	164
InvalidDestCount	Number of queues that failed to open	164
ObjectRecOffset	Offset of first object record from start of MQOD	165
ResponseRecOffset	Offset of first response record from start of MQOD	165
ObjectRecPtr	Address of first object record	166
ResponseRecPtr	Address of first response record	166

The MQOD structure is used to specify an object by name. The following types of object are valid:

- Queue or distribution list
- Namelist (MVS/ESA only)
- Process definition
- Queue manager

The current version of MQOD is MQOD_VERSION_2. Fields that exist only in the version-2 structure are identified as such in the descriptions that follow. The declarations of MQOD provided in the header, COPY, and INCLUDE files for the supported programming languages contain the new fields, but the initial value provided for the *Version* field is MQOD_VERSION_1; this ensures compatibility with existing applications. To use the new fields, the application must set the version number to MQOD_VERSION_2. Applications which are intended to be portable between several environments should use a version-2 MQOD only if all of those environments support version 2.

The version-2 structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

To open a distribution list, a version-2 MQOD must be used.

This structure is an input/output parameter for the MQOPEN and MQPUT1 calls.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQOD_STRUC_ID
Identifier for object descriptor structure.

For the C programming language, the constant MQOD_STRUC_ID_ARRAY is also defined; this has the same value as MQOD_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQOD_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be one of the following:

MQOD_VERSION_1
Version-1 object descriptor structure.

This version is supported in all environments.

MQOD_VERSION_2
Version-2 object descriptor structure.

This version is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields that exist only in the version-2 structure are identified as such in the descriptions that follow.

The following constant specifies the version number of the current version:

MQOD_CURRENT_VERSION
Current version of object descriptor structure.

This is always an input field. The initial value of this field is MQOD_VERSION_1.

ObjectType (MQLONG)
Object type.

Type of object being named in *ObjectName*. Possible values are:

MQOT_Q
Queue.
MQOT_NAMELIST
Namelist (MVS/ESA only).
MQOT_PROCESS
Process definition (not 16-bit Windows, 32-bit Windows).
MQOT_Q_MGR
Queue manager.
MQOT_RESERVED_1
Reserved (MVS/ESA only).

This is always an input field. The initial value of this field is MQOT_Q.

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. For more information about names, see the *MQSeries Application Programming Guide*.

If *ObjectType* is MQOT_Q_MGR, special rules apply; in this case the name must be entirely blank up to the first null character or the end of the field.

If *ObjectName* is the name of a model queue, the queue manager creates a dynamic queue with the attributes of the model queue, and returns in the *ObjectName* field the name of the queue created. A model queue can be specified only for the MQOPEN call.

If a distribution list is being opened (that is, *RecsPresent* is present and greater than zero), *ObjectName* must be blank or the null string. If this condition is not satisfied, the call fails with reason code MQRC_OBJECT_NAME_ERROR.

This is an input/output field for the MQOPEN call when *ObjectName* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ObjectQMgrName (MQCHAR48)
Object queue manager name.

This is 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 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.

If *ObjectType* is MQOT_Q_MGR, the name of the local queue manager must either be specified explicitly, or specified as blank.

If *ObjectName* is the name of a model queue, the queue manager creates a dynamic queue with the attributes of the model queue, and returns in the *ObjectQMgrName* field the name of the queue manager on which the queue is created; this is the name of the local queue manager. A model queue can be specified only for the MQOPEN call.

If a distribution list is being opened (that is, *RecsPresent* is greater than zero), *ObjectQMgrName* must be blank or the null string. If this condition is not satisfied, the call fails with reason code MQRC_OBJECT_Q_MGR_NAME_ERROR.

This is an input/output field for the MQOPEN call when *ObjectName* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial

value of this field is the null string in C, and 48 blank characters in other programming languages.

DynamicQName (MQCHAR48)
Dynamic queue name.

This is an input field that is ignored unless *ObjectName* specifies the name of a model queue. If it does, this field specifies the name of the dynamic queue to be created.

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. A completely blank name (or one in which only blanks appear before the first null character) is not valid if *ObjectName* specifies the name of a model queue.

If the last nonblank character in the name is an asterisk (*), the queue manager replaces the asterisk with a string of characters that guarantees that the name generated for the queue is unique at the local queue manager. To allow a sufficient number of characters for this, the asterisk is valid only in positions 1 through 33. There must be no characters other than blanks or a null character following the asterisk.

It is valid for the asterisk to appear in the first character position, in which case the name consists solely of the characters generated by the queue manager.

On MVS/ESA, it is not recommended to use a name with the asterisk in the first character position, as there can be no security checks made on a queue whose full name is generated automatically.

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is determined by the environment:

- On MVS/ESA, the value is `'CSQ.*'`.
- On other platforms, the value is `'AMQ.*'`.

In all cases, the value is a null-terminated string in C, and a blank-padded string in other programming languages.

AlternateUserId (MQCHAR12)
Alternate user identifier.

If MQOO_ALTERNATE_USER_AUTHORITY is specified for the MQOPEN call, or MQPMO_ALTERNATE_USER_AUTHORITY for the MQPUT1 call, this field contains an alternate user identifier that is to be used to check the authorization for the open, in place of the user identifier that the application is currently running under. Some checks, however, are still carried out with the current user identifier (for example, context checks).

On MVS/ESA, only the first 8 characters of *AlternateUserId* are used to check the authorization for the open. However, the current user identifier must be authorized to specify this particular alternate user identifier; all 12 characters of the alternate user identifier are used for this check. The user ID must contain only characters allowed by the external security manager.

In the following environments, this field is accepted but ignored: 16-bit Windows, 32-bit Windows.

If MQOO_ALTERNATE_USER_AUTHORITY or MQPMO_ALTERNATE_USER_AUTHORITY is specified and this field is entirely blank up to the first null character or the end of the field, the open can succeed only if no user authorization is needed to open this object with the options specified.

If neither MQOO_ALTERNATE_USER_AUTHORITY nor MQPMO_ALTERNATE_USER_AUTHORITY is specified, this field is ignored.

This is an input field. The length of this field is given by MQ_USER_ID_LENGTH. The initial value of this field is the null string in C, and 12 blank characters in other programming languages.

The remaining fields in this structure are not present if *Version* is less than MQOD_VERSION_2.

RecsPresent (MQLONG)
Number of object records present.

This is the number of MQOR object records that have been provided by the application. If this number is greater than zero, it indicates that a distribution list is being opened, with *RecsPresent* being the number of destination queues in the list. It is valid for a distribution list to contain only one destination.

The value of *RecsPresent* must not be less than zero, and if it is greater than zero *ObjectType* must be MQOT_Q; the call fails with reason code MQRC_RECS_PRESENT_ERROR if these conditions are not satisfied.

This is an input field. The initial value of this field is 0. This field is not present if *Version* is less than MQOD_VERSION_2.

KnownDestCount (MQLONG)
Number of local queues opened successfully.

This is the number of queues in the distribution list that resolve to local queues and that were opened successfully. The count does not include queues that resolve to remote queues (even though a local transmission queue is used initially to store the message). If present, this field is also set when opening a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not present if *Version* is less than MQOD_VERSION_2.

UnknownDestCount (MQLONG)
Number of remote queues opened successfully

This is the number of queues in the distribution list that resolve to remote queues and that were opened successfully. If present, this field is also set when opening a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not present if *Version* is less than MQOD_VERSION_2.

InvalidDestCount (MQLONG)
Number of queues that failed to open.

This is the number of queues in the distribution list that failed to open successfully. If present, this field is also set when opening a single queue which is not in a distribution list.

Note: If present, this field is set *only* if the *CompCode* parameter on the MQOPEN or MQPUT1 call is MQCC_OK or MQCC_WARNING; it is *not* set if the *CompCode* parameter is MQCC_FAILED.

This is an output field. The initial value of this field is 0. This field is not present if *Version* is less than MQOD_VERSION_2.

ObjectRecOffset (MQLONG)
Offset of first object record from start of MQOD.

This is the offset in bytes of the first MQOR object record from the start of the MQOD structure. The offset can be positive or negative. *ObjectRecOffset* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

When a distribution list is being opened, an array of one or more MQOR object records must be provided in order to specify the names of the destination queues in the distribution list. This can be done in one of two ways:

- By using the offset field *ObjectRecOffset*

  In this case, the application should declare its own structure containing an MQOD followed by the array of MQOR records (with as many array elements as are needed), and set *ObjectRecOffset* to the offset of the first element in the array from the start of the MQOD. Care must be taken to ensure that this offset is correct.

  Using *ObjectRecOffset* is recommended for programming languages which do not support the pointer data type, or which implement the pointer data type in a fashion which is not portable to different environments (for example, the COBOL programming language).

- By using the pointer field *ObjectRecPtr*

  In this case, the application can declare the array of MQOR structures separately from the MQOD structure, and set *ObjectRecPtr* to the address of the array.

  Using *ObjectRecPtr* is recommended for programming languages which support the pointer data type in a fashion which is portable to different environments (for example, the C programming language).

Whichever technique is chosen, one of *ObjectRecOffset* and *ObjectRecPtr* must be used; the call fails with reason code MQRC_OBJECT_RECORDS_ERROR if both are zero, or both are nonzero.

This is an input field. The initial value of this field is 0. This field is not present if *Version* is less than MQOD_VERSION_2.

ResponseRecOffset (MQLONG)
Offset of first response record from start of MQOD.

This is the offset in bytes of the first MQRR response record from the start of the MQOD structure. The offset can be positive or negative. *ResponseRecOffset* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

When a distribution list is being opened, an array of one or more MQRR response records can be provided in order to identify the queues that

failed to open (*CompCode* field in MQRR), and the reason for each failure (*Reason* field in MQRR). The data is returned in the array of response records in the same order as the queue names occur in the array of object records. The queue manager sets the response records only when the outcome of the call is mixed (that is, some queues were opened successfully while others failed, or all failed but for differing reasons); reason code MQRC_MULTIPLE_REASONS from the call indicates this case. If the same reason code applies to all queues, that reason is returned in the *Reason* parameter of the MQOPEN or MQPUT1 call, and the response records are not set. Response records are optional, but if they are supplied there must be *RecsPresent* of them.

The response records can be provided in the same way as the object records, either by specifying an offset in *ResponseRecOffset*, or by specifying an address in *ResponseRecPtr*; see the description of *ObjectRecOffset* above for details of how to do this. However, no more than one of *ResponseRecOffset* and *ResponseRecPtr* can be used; the call fails with reason code MQRC_RESPONSE_RECORDS_ERROR if both are nonzero.

For the MQPUT1 call, these response records are used to return information about errors that occur when the message is sent to the queues in the distribution list, as well as errors that occur when the queues are opened. The completion code and reason code from the put operation for a queue replace those from the open operation for that queue only if the completion code from the latter was MQCC_OK or MQCC_WARNING.

This is an input field. The initial value of this field is 0. This field is not present if *Version* is less than MQOD_VERSION_2.

ObjectRecPtr (MQPTR)
Address of first object record.

This is the address of the first MQOR object record. *ObjectRecPtr* is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

Either *ObjectRecPtr* or *ObjectRecOffset* can be used to specify the object records, but not both; see the description of the *ObjectRecOffset* field above for details. If *ObjectRecPtr* is not used, it must be set to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is not present if *Version* is less than MQOD_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

ResponseRecPtr (MQPTR)
Address of first response record.

This is the address of the first MQRR response record. *ResponseRecPtr*

is used only when a distribution list is being opened. The field is ignored if *RecsPresent* is zero.

Either *ResponseRecPtr* or *ResponseRecOffset* can be used to specify the response records, but not both; see the description of the *ResponseRecOffset* field above for details. If *ResponseRecPtr* is not used, it must be set to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is not present if *Version* is less than MQOD_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

Table 40 (Page 1 of 2). Initial values of fields in MQOD

Field name	Name of constant	Value of constant
StrucId	MQOD_STRUC_ID	'ODbb' (See note 1)
Version	MQOD_VERSION_1	1
ObjectType	MQOT_Q	1
ObjectName	None	Blanks (See note 2)
ObjectQMgrName	None	Blanks
DynamicQName	None	'CSQ.*' on MVS/ESA; 'AMQ.*' otherwise
AlternateUserId	None	Blanks
RecsPresent	None	0
KnownDestCount	None	0
UnknownDestCount	None	0
InvalidDestCount	None	0
ObjectRecOffset	None	0
ResponseRecOffset	None	0
ObjectRecPtr	None	Null pointer or null bytes

Table 40 (Page 2 of 2). Initial values of fields in MQOD		
Field name	**Name of constant**	**Value of constant**
ResponseRecPtr	None	Null pointer or null bytes

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQOD_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQOD MyOD = {MQOD_DEFAULT};
   ```

C language declaration

```
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;    /* Alternate user identifier */
  MQLONG    RecsPresent;        /* Number of object records present */
  MQLONG    KnownDestCount;     /* Number of local queues opened suc-
                                   cessfully */
  MQLONG    UnknownDestCount;   /* Number of remote queues opened suc-
                                   cessfully */
  MQLONG    InvalidDestCount;   /* Number of queues that failed to
                                   open */
  MQLONG    ObjectRecOffset;    /* Offset of first object record from
                                   start of MQOD */
  MQLONG    ResponseRecOffset;  /* Offset of first response record from
                                   start of MQOD */
  MQPTR     ObjectRecPtr;       /* Address of first object record */
  MQPTR     ResponseRecPtr;     /* Address of first response record */
 } MQOD;
```

COBOL language declaration

```
**   MQOD structure
  10 MQOD.
**    Structure identifier
   15 MQOD-STRUCID           PIC X(4).
**    Structure version number
   15 MQOD-VERSION           PIC S9(9) BINARY.
**    Object type
   15 MQOD-OBJECTTYPE        PIC S9(9) BINARY.
**    Object name
   15 MQOD-OBJECTNAME        PIC X(48).
**    Object queue manager name
   15 MQOD-OBJECTQMGRNAME    PIC X(48).
**    Dynamic queue name
   15 MQOD-DYNAMICQNAME      PIC X(48).
**    Alternate user identifier
```

```
   15 MQOD-ALTERNATEUSERID   PIC X(12).
**    Number of object records present
   15 MQOD-RECSPRESENT       PIC S9(9) BINARY.
**    Number of local queues opened successfully
   15 MQOD-KNOWNDESTCOUNT    PIC S9(9) BINARY.
**    Number of remote queues opened successfully
   15 MQOD-UNKNOWNDESTCOUNT  PIC S9(9) BINARY.
**    Number of queues that failed to open
   15 MQOD-INVALIDDESTCOUNT  PIC S9(9) BINARY.
**    Offset of first object record from start of MQOD
   15 MQOD-OBJECTRECOFFSET   PIC S9(9) BINARY.
**    Offset of first response record from start of MQOD
   15 MQOD-RESPONSERECOFFSET PIC S9(9) BINARY.
**    Address of first object record
   15 MQOD-OBJECTRECPTR      POINTER.
**    Address of first response record
   15 MQOD-RESPONSERECPTR    POINTER.
```

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

```
dcl
 1 MQOD based,
  3 StrucId           char(4),       /* Structure identifier */
  3 Version           fixed bin(31), /* Structure version number */
  3 ObjectType        fixed bin(31), /* Object type */
  3 ObjectName        char(48),      /* Object name */
  3 ObjectQMgrName    char(48),      /* Object queue manager name */
  3 DynamicQName      char(48),      /* Dynamic queue name */
  3 AlternateUserId   char(12),      /* Alternate user identifier */
  3 RecsPresent       fixed bin(31), /* Number of object records
                                        present */
  3 KnownDestCount    fixed bin(31), /* Number of local queues opened
                                        successfully */
  3 UnknownDestCount  fixed bin(31), /* Number of remote queues opened
                                        successfully */
  3 InvalidDestCount  fixed bin(31), /* Number of queues that failed to
                                        open */
  3 ObjectRecOffset   fixed bin(31), /* Offset of first object record
                                        from start of MQOD */
  3 ResponseRecOffset fixed bin(31), /* Offset of first response record
                                        from start of MQOD */
  3 ObjectRecPtr      pointer,       /* Address of first object
                                        record */
  3 ResponseRecPtr    pointer;       /* Address of first response
                                        record */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQOD                        DSECT
MQOD_STRUCID                DS   CL4      Structure identifier
MQOD_VERSION                DS   F        Structure version number
MQOD_OBJECTTYPE             DS   F        Object type
MQOD_OBJECTNAME             DS   CL48     Object name
MQOD_OBJECTQMGRNAME         DS   CL48     Object queue manager name
MQOD_DYNAMICQNAME           DS   CL48     Dynamic queue name
MQOD_ALTERNATEUSERID        DS   CL12     Alternate user identifier
MQOD_LENGTH                 EQU  *-MQOD   Length of structure
```

```
                                        ORG  MQOD
MQOD_AREA                               DS   CL(MQOD_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQOD^DEF (*);BEGINSTRUCT        STRUCID;
BEGIN STRING BYTE [0:3]; END;INT(32)       VERSION;
INT(32)       OBJECTTYPE;STRUCT
   OBJECTNAME;
BEGIN STRING BYTE [0:47]; END;STRUCT        OBJECTQMGRNAME;
BEGIN STRING BYTE [0:47]; END;STRUCT        DYNAMICQNAME;
BEGIN STRING BYTE [0:47]; END;STRUCT        ALTERNATEUSERID;
BEGIN STRING BYTE [0:11]; END;
```

MQOR – Object record

The following table summarizes the fields in the structure.

Table 41. Fields in MQOR

Field	Description	Page
ObjectName	Object name	171
ObjectQMgrName	Object queue manager name	171

The MQOR structure is used to specify the queue name and queue-manager name of a single destination queue. By providing an array of these structures on the MQOPEN call, it is possible to open a list of queues; this list is called a *distribution list*. Each message put using the queue handle returned by that MQOPEN call is placed on each of the queues in the list, provided that the queue was opened successfully.

The character data in the MQOR structure must be in the queue-manager's character set. MQOR is an input structure for the MQOPEN and MQPUT1 calls.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields

ObjectName (MQCHAR48)
Object name.

This is the same as the *ObjectName* field in the MQOD structure (see MQOD for details), except that:

- It must be the name of a queue.
- It must not be the name of a model queue.

This is always an input field. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ObjectQMgrName (MQCHAR48)
Object queue manager name.

This is the same as the *ObjectQMgrName* field in the MQOD structure (see MQOD for details).

This is always an input field. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

Table 42. Initial values of fields in MQOR

Field name	Name of constant	Value of constant
ObjectName	None	Blanks (See note 1)
ObjectQMgrName	None	Blanks

Notes:

1. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
2. In the C programming language, the macro variable MQOR_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQOR MyOR = {MQOR_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQOR {
  MQCHAR48  ObjectName;      /* Object name */
  MQCHAR48  ObjectQMgrName;  /* Object queue manager name */
 } MQOR;
```

COBOL language declaration

```
**   MQOR structure
  10 MQOR.
**    Object name
   15 MQOR-OBJECTNAME     PIC X(48).
**    Object queue manager name
   15 MQOR-OBJECTQMGRNAME PIC X(48).
```

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

```
dcl
 1 MQOR based,
  3 ObjectName     char(48), /* Object name */
  3 ObjectQMgrName char(48); /* Object queue manager name */
```

MQPMO – Put message options

The following table summarizes the fields in the structure.

Table 43. Fields in MQPMO

Field	Description	Page
StrucId	Structure identifier	174
Version	Structure version number	174
Options	Options that control the action of MQPUT and MQPUT1	174
Context	Object handle of input queue	184
KnownDestCount	Number of messages sent successfully to local queues	184
UnknownDestCount	Number of messages sent successfully to remote queues	185
InvalidDestCount	Number of messages that could not be sent	185
ResolvedQName	Resolved name of destination queue	185
ResolvedQMgrName	Resolved name of destination queue manager	185
Note: The remaining fields are supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.		
RecsPresent	Number of put message records or response records present	186
PutMsgRecFields	Flags indicating which MQPMR fields are present	186
PutMsgRecOffset	Offset of first put-message record from start of MQPMO	187
ResponseRecOffset	Offset of first response record from start of MQPMO	188
PutMsgRecPtr	Address of first put message record	189
ResponseRecPtr	Address of first response record	189

The current version of MQPMO is MQPMO_VERSION_2. Fields that exist only in the version-2 structure are identified as such in the descriptions that follow. The declarations of MQPMO provided in the header, COPY, and INCLUDE files for the supported programming languages contain the new fields, but the initial value provided for the *Version* field is MQPMO_VERSION_1; this ensures compatibility with existing applications. To use the new fields, the application must set the version number to MQPMO_VERSION_2. Applications which are intended to be portable between several environments should use a version-2 MQPMO only if all of those environments support version 2.

The version-2 structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

The MQPMO structure is an input/output parameter for the MQPUT and MQPUT1 calls.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQPMO_STRUC_ID
Identifier for put-message options structure.

For the C programming language, the constant MQPMO_STRUC_ID_ARRAY is also defined; this has the same value as MQPMO_STRUC_ID, but is an array of characters instead of a string.

This is always an input field. The initial value of this field is MQPMO_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be one of the following:

MQPMO_VERSION_1
Version-1 put-message options structure.

This version is supported in all environments.

MQPMO_VERSION_2
Version-2 put-message options structure.

This version is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields that exist only in the version-2 structure are identified as such in the descriptions that follow.

The following constant specifies the version number of the current version:

MQPMO_CURRENT_VERSION
Current version of put-message options structure.

This is always an input field. The initial value of this field is MQPMO_VERSION_1.

Options (MQLONG)
Options that control the action of MQPUT and MQPUT1.

Any or none of the following can be specified. If more than one is required the values can be:

- Added together (do not add the same constant more than once), or
- Combined using the bitwise OR operation (if the programming language supports bit operations).

Combinations that are not valid are noted; any other combinations are valid. The following options are described:

MQPMO_SYNCPOINT
MQPMO_NO_SYNCPOINT
MQPMO_NEW_MSG_ID

MQPMO_NEW_CORREL_ID
MQPMO_LOGICAL_ORDER
MQPMO_NO_CONTEXT
MQPMO_DEFAULT_CONTEXT
MQPMO_PASS_IDENTITY_CONTEXT
MQPMO_PASS_ALL_CONTEXT
MQPMO_SET_IDENTITY_CONTEXT
MQPMO_SET_ALL_CONTEXT
MQPMO_ALTERNATE_USER_AUTHORITY
MQPMO_FAIL_IF_QUIESCING
MQPMO_NONE

MQPMO_SYNCPOINT
Put message with syncpoint control.

The request is to operate within the normal unit of work protocols. The message is not visible outside the unit of work until the unit of work is committed. If the unit of work is backed out, the message is deleted.

If neither this option nor MQPMO_NO_SYNCPOINT is specified, the inclusion of the put request in unit of work protocols is determined by the environment:

- On MVS/ESA, the put request is within a unit of work.
- In all other environments, the put request is not within a unit of work.

Because of these differences, an application which is intended to be portable should not allow this option to default; either MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT should be specified explicitly.

MQPMO_SYNCPOINT must *not* be specified with MQPMO_NO_SYNCPOINT.

MQPMO_NO_SYNCPOINT
Put message without syncpoint control.

The request is to operate outside the normal unit of work protocols. The message is available immediately, and it cannot be deleted by backing out a unit of work.

If neither this option nor MQPMO_SYNCPOINT is specified, the inclusion of the put request in unit of work protocols is determined by the environment:

- On MVS/ESA, the put request is within a unit of work.
- In all other environments, the put request is not within a unit of work.

Because of these differences, an application which is intended to be portable should not allow this option to default; either MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT should be specified explicitly.

MQPMO_NO_SYNCPOINT must *not* be specified with MQPMO_SYNCPOINT.

| On Tandem NSK, if MQPUT is issued outside a Tandem TMF
| transaction *without* the MQPMO_NO_SYNCPOINT option, the
| reason code MQRC_UNIT_OF_WORK_NOT_STARTED is returned.

MQPMO_NEW_MSG_ID
Generate a new message identifier.

This option causes the queue manager to replace the contents of the *MsgId* field in MQMD with a new message identifier. This message identifier is sent with the message, and returned to the application on output from the MQPUT or MQPUT1 call.

This option can also be specified when the message is being put to a distribution list; see the description of the *MsgId* field in the MQPMR structure for details.

Using this option relieves the application of the need to reset the *MsgId* field to MQMI_NONE prior to each MQPUT or MQPUT1 call.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQPMO_NEW_CORREL_ID
Generate a new correlation identifier.

This option causes the queue manager to replace the contents of the *CorrelId* field in MQMD with a new correlation identifier. This correlation identifier is sent with the message, and returned to the application on output from the MQPUT or MQPUT1 call.

This option can also be specified when the message is being put to a distribution list; see the description of the *CorrelId* field in the MQPMR structure for details.

MQPMO_NEW_CORREL_ID is useful in situations where the application requires a unique correlation identifier.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Group and segment option: The option described below relates to messages in groups and segments of logical messages. The following definitions may be of help in understanding this option:

Physical message
This is the smallest unit of information that can be placed on or removed from a queue; it often corresponds to the information specified or retrieved on a single MQPUT, MQPUT1, or MQGET call. Every physical message has its own message descriptor (MQMD). Generally, physical messages are distinguished by differing values for the message identifier (*MsgId* field in MQMD), although this is not enforced by the queue manager.

Logical message
This is a single unit of application information. In the absence of system constraints, a logical message would be the same as a physical message. But where logical messages are extremely large, system constraints may make it advisable or necessary to split a

logical message into two or more physical messages, called *segments*.

A logical message that has been segmented consists of two or more physical messages that have the same nonnull group identifier (*GroupId* field in MQMD), and the same message sequence number (*MsgSeqNumber* field in MQMD). The segments are distinguished by differing values for the segment offset (*Offset* field in MQMD), which gives the offset of the data in the physical message from the start of the data in the logical message. Because each segment is a physical message, the segments in a logical message usually have differing message identifiers.

A logical message that has not been segmented, but for which segmentation has been permitted by the sending application, also has a nonnull group identifier, although in this case there is only one physical message with that group identifier if the logical message does not belong to a message group. Logical messages for which segmentation has been inhibited by the sending application have a null group identifier (MQGI_NONE), unless the logical message belongs to a message group.

Message group

This is a set of one or more logical messages that have the same nonnull group identifier. The logical messages in the group are distinguished by differing values for the message sequence number, which is an integer in the range 1 through n, where n is the number of logical messages in the group. If one or more of the logical messages is segmented, there will be more than n physical messages in the group.

MQPMO_LOGICAL_ORDER

Messages in groups and segments of logical messages will be put in logical order.

This option tells the queue manager how the application will put messages in groups and segments of logical messages. It can be specified only on the MQPUT call; it is *not* valid on the MQPUT1 call.

If MQPMO_LOGICAL_ORDER is specified, it indicates that the application will use successive MQPUT calls to:

- Put the segments in each logical message in the order of increasing segment offset, starting from 0, with no gaps.
- Put all of the segments in one logical message before putting the segments in the next logical message.
- Put the logical messages in each message group in the order of increasing message sequence number, starting from 1, with no gaps.
- Put all of the logical messages in one message group before putting logical messages in the next message group.

The above order is called “logical order”.

Because the application has told the queue manager how it will put messages in groups and segments of logical messages, the application does not have to maintain and update the group and

segment information on each MQPUT call, as the queue manager does this. Specifically, it means that the application does not need to set the *GroupId*, *MsgSeqNumber*, and *Offset* fields in MQMD, as the queue manager sets these to the appropriate values. The application need set only the the *MsgFlags* field in MQMD, to indicate when messages belong to groups or are segments of logical messages, and to indicate the last message in a group or last segment of a logical message.

Once a message group or logical message has been started, subsequent MQPUT calls must specify the appropriate MQMF_★ flags in *MsgFlags* in MQMD. If the application tries to put a message not in a group when there is an unterminated message group, or put a message which is not a segment when there is an unterminated logical message, the call fails with reason code MQRC_INCOMPLETE_GROUP or MQRC_INCOMPLETE_MSG, as appropriate. However, the queue manager retains the information about the current message group and/or current logical message, and the application can terminate them by sending a message (possibly with no application message data) specifying MQMF_LAST_MSG_IN_GROUP and/or MQMF_LAST_SEGMENT as appropriate, before reissuing the MQPUT call to put the message that is not in the group or not a segment.

Table 44 on page 179 shows the combinations of options and flags that are valid, and the values of the *GroupId*, *MsgSeqNumber*, and *Offset* fields that the queue manager uses in each case. Combinations of options and flags that are not shown in the table are not valid. The abbreviated column headings denote the following options, flags, and group and logical-message status:

- **LOG ORD** means the MQPMO_LOGICAL_ORDER option.
- **MIG** means the MQMF_MSG_IN_GROUP and/or MQMF_LAST_MSG_IN_GROUP flag.
- **SEG** means the MQMF_SEGMENT and/or MQMF_LAST_SEGMENT flag.
- **SEG OK** means the MQMF_SEGMENTATION_ALLOWED flag.
- **Cur grp** means that a current message group exists prior to the call.
- **Cur log msg** means that a current logical message exists prior to the call.

In the table:

- "(√)" indicates that the row applies whether or not there is a √ in that column.
- "Previous" denotes the value used for that field in the previous message for the queue handle.

Table 44. MQPUT options relating to messages in groups and segments of logical messages

Options you specify				Group and log-msg status prior to call		Values the queue manager uses		
LOG ORD	**MIG**	**SEG**	**SEG OK**	**Cur grp**	**Cur log msg**	*GroupId*	*MsgSeqNumber*	*Offset*
√						MQGI_NONE	1	0
√			√			New group id	1	0
√		√	(√)			New group id	1	0
√		√	(√)		√	Previous group id	1	Previous offset + previous segment length
√	√	(√)	(√)			New group id	1	0
√	√	(√)	(√)	√		Previous group id	Previous sequence number + 1	0
√	√	√	(√)	√	√	Previous group id	Previous sequence number	Previous offset + previous segment length
				(√)	(√)	MQGI_NONE	1	0
			√	(√)	(√)	New group id if MQGI_NONE, else value in field	1	0
		√	(√)	(√)	(√)	New group id if MQGI_NONE, else value in field	1	Value in field
	√		(√)	(√)	(√)	New group id if MQGI_NONE, else value in field	Value in field	0
	√	√	(√)	(√)	(√)	New group id if MQGI_NONE, else value in field	Value in field	Value in field

Notes:

- MQPMO_LOGICAL_ORDER is not valid on the MQPUT1 call.
- For the *MsgId* field, the queue manager generates a new message identifier if MQPMO_NEW_MSG_ID or MQMI_NONE is specified, and uses the value in the field otherwise.
- For the *CorrelId* field, the queue manager generates a new correlation identifier if MQPMO_NEW_CORREL_ID is specified, and uses the value in the field otherwise.

When MQPMO_LOGICAL_ORDER is specified, the queue manager requires that all messages in a group and segments in a logical message be put with the same value in the *Persistence* field in MQMD, that is, all must be persistent, or all must be nonpersistent. If this condition is not satisfied, the MQPUT call fails with reason code MQRC_INCONSISTENT_PERSISTENCE.

The MQPMO_LOGICAL_ORDER option affects units of work as follows:

- If the first physical message in a group or logical message is put within a unit of work, all of the other physical messages in the group or logical message must be put within a unit of work, if the same queue handle is used. However, they need not be put within the *same* unit of work. This allows a message group or logical message consisting of many physical messages to be split across two or more consecutive units of work for the queue handle.

- If the first physical message in a group or logical message is *not* put within a unit of work, none of the other physical messages in the group or logical message can be put within a unit of work, if the same queue handle is used.

If these conditions are not satisfied, the MQPUT call fails with reason code MQRC_INCONSISTENT_UOW.

When MQPMO_LOGICAL_ORDER is specified, the MQMD supplied on the MQPUT call must not be less than MQMD_VERSION_2. If this condition is not satisfied, the call fails with reason code MQRC_WRONG_MD_VERSION.

If MQPMO_LOGICAL_ORDER is *not* specified, messages in groups and segments of logical messages can be put in any order, and it is not necessary to put complete message groups or complete logical messages. It is the application's responsibility to ensure that the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields have appropriate values.

This is the technique that can be used to restart a message group or logical message in the middle, after a system failure has occurred. When the system restarts, the application can set the *GroupId*, *MsgSeqNumber*, *Offset*, *MsgFlags*, and *Persistence* fields to the appropriate values, and then issue the MQPUT call with MQPMO_SYNCPOINT or MQPMO_NO_SYNCPOINT set as desired, but *without* specifying MQPMO_LOGICAL_ORDER. If this call is successful, the queue manager retains the group and segment information, and subsequent MQPUT calls using that queue handle can specify MQPMO_LOGICAL_ORDER as normal.

The group and segment information that the queue manager retains for the MQPUT call is separate from the group and segment information that it retains for the MQGET call.

For any given queue handle, the application is free to mix MQPUT calls that specify MQPMO_LOGICAL_ORDER with MQPUT calls that do not, but the following points should be noted:

- Each successful MQPUT call that does *not* specify MQPMO_LOGICAL_ORDER causes the queue manager to set the group and segment information for the queue handle to the values specified by the application; this replaces the existing group and segment information retained by the queue manager for the queue handle.
- If MQPMO_LOGICAL_ORDER is *not* specified, the call does not fail if there is a current message group or logical message, but the message or segment put is not the next one in the group or logical message. The call may however succeed with an MQCC_WARNING completion code. Table 45 on page 181 shows the various cases that can arise. In these cases, if the completion code is not MQCC_OK, the reason code is one of the following (as appropriate):

  MQRC_INCOMPLETE_GROUP
  MQRC_INCOMPLETE_MSG
  MQRC_INCONSISTENT_PERSISTENCE
  MQRC_INCONSISTENT_UOW

Note: The queue manager does not check the group and segment information for the MQPUT1 call.

Table 45. Outcome when MQPUT or MQCLOSE call not consistent with group and segment information

Current call	Previous call	
	MQPUT with MQPMO_LOGICAL_ORDER	MQPUT without MQPMO_LOGICAL_ORDER
MQPUT with MQPMO_LOGICAL_ORDER	MQCC_FAILED	MQCC_FAILED
MQPUT without MQPMO_LOGICAL_ORDER	MQCC_WARNING	MQCC_OK
MQCLOSE with an unterminated group or logical message	MQCC_WARNING	MQCC_OK

Applications that simply want to put messages and segments in logical order are recommended to specify MQPMO_LOGICAL_ORDER, as this is the simplest option to use. This option relieves the application of the need to manage the group and segment information, because the queue manager manages that information. However, specialized applications may need more control than provided by the MQPMO_LOGICAL_ORDER option, and this can be achieved by not specifying that option. If this is done, the application must ensure that the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD are set correctly, prior to each MQPUT or MQPUT1 call.

For example, an application that wants to *forward* physical messages that it receives, without regard for whether those messages are in groups or segments of logical messages, should *not* specify MQPMO_LOGICAL_ORDER. There are two reasons for this:

- If the messages are retrieved and put in order, specifying MQPMO_LOGICAL_ORDER will cause a new group identifier to be assigned to the messages, and this may make it difficult or impossible for the originator of the messages to correlate any reply or report messages that result from the message group.
- In a complex network with multiple paths between sending and receiving queue managers, the physical messages may arrive out of order. By specifying neither MQPMO_LOGICAL_ORDER, nor the corresponding MQGMO_LOGICAL_ORDER on the MQGET call, the forwarding application can retrieve and forward each physical message as soon as it arrives, without having to wait for the next one in logical order to arrive.

Applications that generate report messages for messages in groups or segments of logical messages should also not specify MQPMO_LOGICAL_ORDER when putting the report message.

MQPMO_LOGICAL_ORDER can be specified with any of the other MQPMO_★ options.

This option is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQPMO_NO_CONTEXT
No context is to be associated with the message.

Both identity and origin context are set to indicate no context. This means that the context fields in MQMD are set to:

- Blanks for character fields
- Nulls for byte fields
- Zeros for numeric fields

MQPMO_DEFAULT_CONTEXT
Use default context.

The message is to have default context information associated with it, for both identity and origin. The queue manager sets the context fields in the message descriptor as follows:

Field in MQMD	Value used
UserIdentifier	Determined from the environment if possible; set to blanks otherwise.
AccountingToken	Determined from the environment if possible; set to MQACT_NONE otherwise.
ApplIdentityData	Set to blanks.
PutApplType	Determined from the environment.
PutApplName	Determined from the environment if possible; set to blanks otherwise.
PutDate	Set to date when message is put.
PutTime	Set to time when message is put.
ApplOriginData	Set to blanks.

For more information on message context, see the *MQSeries Application Programming Guide*.

This is the default action if no context options are specified.

MQPMO_PASS_IDENTITY_CONTEXT
Pass identity context from an input queue handle.

The message is to have context information associated with it. Identity context is taken from the queue handle specified in the *Context* field. Origin context information is generated by the queue manager in the same way that it is for MQPMO_DEFAULT_CONTEXT (see above for values). For more information on message context, see the *MQSeries Application Programming Guide*.

For the MQPUT call, the queue must have been opened with the MQOO_PASS_IDENTITY_CONTEXT option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_PASS_IDENTITY_CONTEXT option.

This option is not supported in the following environments: 16-bit Windows, 32-bit Windows.

MQPMO_PASS_ALL_CONTEXT
Pass all context from an input queue handle.

The message is to have context information associated with it. Both identity and origin context are taken from the queue handle specified

in the *Context* field. For more information on message context, see the *MQSeries Application Programming Guide*.

For the MQPUT call, the queue must have been opened with the MQOO_PASS_ALL_CONTEXT option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_PASS_ALL_CONTEXT option.

This option is not supported in the following environments: 16-bit Windows, 32-bit Windows.

MQPMO_SET_IDENTITY_CONTEXT
Set identity context from the application.

The message is to have context information associated with it. The application specifies the identity context in the MQMD structure. Origin context information is generated by the queue manager in the same way that it is for MQPMO_DEFAULT_CONTEXT (see above for values). For more information on message context, see the *MQSeries Application Programming Guide*.

For the MQPUT call, the queue must have been opened with the MQOO_SET_IDENTITY_CONTEXT option (or an option that implies it). For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_SET_IDENTITY_CONTEXT option.

MQPMO_SET_ALL_CONTEXT
Set all context from the application.

The message is to have context information associated with it. The application specifies the identity and origin context in the MQMD structure. For more information on message context, see the *MQSeries Application Programming Guide*.

For the MQPUT call, the queue must have been opened with the MQOO_SET_ALL_CONTEXT option. For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the MQOO_SET_ALL_CONTEXT option.

Only one of the MQPMO_*_CONTEXT context options can be specified. If none of these options is specified, MQPMO_DEFAULT_CONTEXT is assumed.

MQPMO_ALTERNATE_USER_AUTHORITY
Validate with specified user identifier.

This indicates that the *AlternateUserId* field in the *ObjDesc* parameter of the MQPUT1 call contains a user identifier that is to be used to validate authority to put messages on the queue. The call can succeed only if this *AlternateUserId* is authorized to open the queue with the specified options, regardless of whether the user identifier under which the application is running is authorized to do so. (This does not apply to the context options specified, however, which are always checked against the user identifier under which the application is running.)

This option is valid only with the MQPUT1 call.

This option is accepted but ignored in the following environments: 16-bit Windows, 32-bit Windows.

MQPMO_FAIL_IF_QUIESCING
Fail if queue manager is quiescing.

This option forces the MQPUT or MQPUT1 call to fail if the queue manager is in the quiescing state.

On MVS/ESA, this option also forces the MQPUT or MQPUT1 call to fail if the connection (for a CICS or IMS application) is in the quiescing state.

The call returns completion code MQCC_FAILED with reason code MQRC_Q_MGR_QUIESCING or MQRC_CONNECTION_QUIESCING.

This option is accepted but ignored in the following environments: 16-bit Windows, 32-bit Windows.

MQPMO_NONE
No options specified.

This value can be used to indicate that no other options have been specified; all options assume their default values. MQPMO_NONE is defined to aid program documentation; it is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

This is an input field. The initial value of the *Options* field is MQPMO_NONE.

Timeout (MQLONG)
Reserved.

This is a reserved field; its value is not significant. The initial value of this field is –1.

Context (MQHOBJ)
Object handle of input queue.

If MQPMO_PASS_IDENTITY_CONTEXT or MQPMO_PASS_ALL_CONTEXT is specified, this field must contain the input queue handle from which context information to be associated with the message being put is taken.

If neither MQPMO_PASS_IDENTITY_CONTEXT nor MQPMO_PASS_ALL_CONTEXT is specified, this field is ignored.

This is an input field. The initial value of this field is 0.

KnownDestCount (MQLONG)
Number of messages sent successfully to local queues.

This is the number of messages that the current MQPUT or MQPUT1 call has sent successfully to queues in the distribution list that are local queues. The count does not include messages sent to queues that resolve to remote queues (even though a local transmission queue is used initially to store the message). This field is also set when putting a message to a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not set if *Version* is less than MQPMO_VERSION_2.

UnknownDestCount (MQLONG)
Number of messages sent successfully to remote queues.

This is the number of messages that the current MQPUT or MQPUT1 call has sent successfully to queues in the distribution list that resolve to remote queues. Messages that the queue manager retains temporarily in distribution-list form count as the number of individual destinations that those distribution lists contain. This field is also set when putting a message to a single queue which is not in a distribution list.

This is an output field. The initial value of this field is 0. This field is not set if *Version* is less than MQPMO_VERSION_2.

InvalidDestCount (MQLONG)
Number of messages that could not be sent.

This is the number of messages that could not be sent to queues in the distribution list. The count includes queues that failed to open, as well as queues that were opened successfully but for which the put operation failed. This field is also set when putting a message to a single queue which is not in a distribution list.

Note: This field is set *only* if the *CompCode* parameter on the MQPUT or MQPUT1 call is MQCC_OK or MQCC_WARNING; it is *not* set if the *CompCode* parameter is MQCC_FAILED.

This is an output field. The initial value of this field is 0. This field is not set if *Version* is less than MQPMO_VERSION_2.

ResolvedQName (MQCHAR48)
Resolved name of destination queue.

This is an output field that is set by the queue manager to the name of the queue (after alias resolution) on which the message will be placed. This can be either the name of a local queue, or the name of a remote queue. If the destination queue opened was a model queue, the name of the dynamic local queue that was created is returned. In all cases, the name returned is the name of a queue that is defined on the queue manager identified by *ResolvedQMgrName*.

If the MQPUT or MQPUT1 call is used to put the message to a distribution list, the value returned in this field is undefined.

This is an output field. The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ResolvedQMgrName (MQCHAR48)
Resolved name of destination queue manager.

This is the name of the queue manager (after alias resolution) that owns the queue specified by *ResolvedQName*.

If the MQPUT or MQPUT1 call is used to put the message to a distribution list, the value returned in this field is undefined.

This is an output field. The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

The remaining fields in this structure are not present if *Version* is less than MQPMO_VERSION_2.

RecsPresent (MQLONG)
Number of put message records or response records present.

This is the number of MQPMR put message records or MQRR response records that have been provided by the application. This number can be greater than zero only if the message is being put to a distribution list. Put message records and response records are optional – the application need not provide any records, or it can choose to provide records of only one type. However, if the application provides records of both types, it must provide *RecsPresent* records of each type.

The value of *RecsPresent* need not be the same as the number of destinations in the distribution list. If too many records are provided, the excess are not used; if too few records are provided, default values are used for the message properties for those destinations that do not have put message records (see *PutMsgRecOffset* below).

If *RecsPresent* is less than zero, or is greater than zero but the message is not being put to a distribution list, the call fails with reason code MQRC_RECS_PRESENT_ERROR.

This is an input field. The initial value of this field is 0. This field is not present if *Version* is less than MQPMO_VERSION_2.

PutMsgRecFields (MQLONG)
Flags indicating which MQPMR fields are present.

This field contains flags that must be set to indicate which MQPMR fields are present in the put message records provided by the application. *PutMsgRecFields* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero, or both *PutMsgRecOffset* and *PutMsgRecPtr* are zero.

For fields that are present, the queue manager uses for each destination the values from the fields in the corresponding put message record. For fields that are absent, the queue manager uses the values from the MQMD structure.

One or more of the following flags can be specified to indicate which fields are present in the put message records:

MQPMRF_MSG_ID
Message-identifier field is present.

MQPMRF_CORREL_ID
Correlation-identifier field is present.

MQPMRF_GROUP_ID
Group-identifier field is present.

MQPMRF_FEEDBACK
Feedback field is present.

MQPMRF_ACCOUNTING_TOKEN
Accounting-token field is present.

If this flag is specified, either MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT must be specified in the *Options* field; if this condition is not satisfied, the call fails with reason code MQRC_PMO_RECORD_FLAGS_ERROR.

If no MQPMR fields are present, the following can be specified:

MQPMRF_NONE
No put-message record fields are present.

If this value is specified, either *RecsPresent* must be zero, or both *PutMsgRecOffset* and *PutMsgRecPtr* must be zero.

MQPMRF_NONE is defined to aid program documentation. It is not intended that this constant be used with any other, but as its value is zero, such use cannot be detected.

If *PutMsgRecFields* contains flags which are not valid, or put message records are provided but *PutMsgRecFields* has the value MQPMRF_NONE, the call fails with reason code MQRC_PMO_RECORD_FLAGS_ERROR.

This is an input field. The initial value of this field is MQPMRF_NONE. This field is not present if *Version* is less than MQPMO_VERSION_2.

PutMsgRecOffset (MQLONG)
Offset of first put message record from start of MQPMO.

This is the offset in bytes of the first MQPMR put message record from the start of the MQPMO structure. The offset can be positive or negative. *PutMsgRecOffset* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

When the message is being put to a distribution list, an array of one or more MQPMR put message records can be provided in order to specify certain properties of the message for each destination individually; these properties are:

- message identifier
- correlation identifier
- group identifier
- feedback value
- accounting token

It is not necessary to specify all of these properties, but whatever subset is chosen, the fields must be specified in the correct order. See the description of the MQPMR structure for further details.

Usually, there should be as many put message records as there are object records specified by MQOD when the distribution list is opened; each put message record supplies the message properties for the queue identified by the corresponding object record. Queues in the distribution list which fail to open must still have put message records allocated for them at the appropriate positions in the array, although the message properties are ignored in this case.

It is possible for the number of put message records to differ from the number of object records. If there are fewer put message records than object records, the message properties for the destinations which do not have put message records are taken from the corresponding fields in the message descriptor MQMD. If there are more put message records than object records, the excess are not used (although it must still be possible to access them). Put message records are optional, but if they are supplied there must be *RecsPresent* of them.

The put message records can be provided in a similar way to the object records in MQOD, either by specifying an offset in *PutMsgRecOffset*, or by specifying an address in *PutMsgRecPtr*; for details of how to do this, see the *ObjectRecOffset* field described in "MQOD – Object descriptor" on page 160.

No more than one of *PutMsgRecOffset* and *PutMsgRecPtr* can be used; the call fails with reason code MQRC_PUT_MSG_RECORDS_ERROR if both are nonzero.

This is an input field. The initial value of this field is 0. This field is not present if *Version* is less than MQPMO_VERSION_2.

ResponseRecOffset (MQLONG)
Offset of first response record from start of MQPMO.

This is the offset in bytes of the first MQRR response record from the start of the MQPMO structure. The offset can be positive or negative. *ResponseRecOffset* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

When the message is being put to a distribution list, an array of one or more MQRR response records can be provided in order to identify the queues to which the message was not sent successfully (*CompCode* field in MQRR), and the reason for each failure (*Reason* field in MQRR). The message may not have been sent either because the queue failed to open, or because the put operation failed. The queue manager sets the response records only when the outcome of the call is mixed (that is, some messages were sent successfully while others failed, or all failed but for differing reasons); reason code MQRC_MULTIPLE_REASONS from the call indicates this case. If the same reason code applies to all queues, that reason is returned in the *Reason* parameter of the MQPUT or MQPUT1 call, and the response records are not set.

Usually, there should be as many response records as there are object records specified by MQOD when the distribution list is opened; when necessary, each response record is set to the completion code and reason code for the put to the queue identified by the corresponding object record. Queues in the distribution list which fail to open must still have response records allocated for them at the appropriate positions in the array, although they are set to the completion code and reason code resulting from the open operation, rather than the put operation.

It is possible for the number of response records to differ from the number of object records. If there are fewer response records than object records, it may not be possible for the application to identify all of the destinations for which the put operation failed, or the reasons for the failures. If there are more response records than object records, the excess are not used

(although it must still be possible to access them). Response records are optional, but if they are supplied there must be *RecsPresent* of them.

The response records can be provided in a similar way to the object records in MQOD, either by specifying an offset in *ResponseRecOffset*, or by specifying an address in *ResponseRecPtr*; for details of how to do this, see the *ObjectRecOffset* field described in "MQOD – Object descriptor" on page 160. However, no more than one of *ResponseRecOffset* and *ResponseRecPtr* can be used; the call fails with reason code MQRC_RESPONSE_RECORDS_ERROR if both are nonzero.

For the MQPUT1 call, this field must be zero. This is because the response information (if requested) is returned in the response records specified by the object descriptor MQOD.

This is an input field. The initial value of this field is 0. This field is not present if *Version* is less than MQPMO_VERSION_2.

PutMsgRecPtr (MQPTR)
Address of first put message record.

This is the address of the first MQPMR put message record. *PutMsgRecPtr* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

Either *PutMsgRecPtr* or *PutMsgRecOffset* can be used to specify the put message records, but not both; see the description of the *PutMsgRecOffset* field above for details. If *PutMsgRecPtr* is not used, it must be set to the null pointer or null bytes.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is not present if *Version* is less than MQPMO_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

ResponseRecPtr (MQPTR)
Address of first response record.

This is the address of the first MQRR response record. *ResponseRecPtr* is used only when the message is being put to a distribution list. The field is ignored if *RecsPresent* is zero.

Either *ResponseRecPtr* or *ResponseRecOffset* can be used to specify the response records, but not both; see the description of the *ResponseRecOffset* field above for details. If *ResponseRecPtr* is not used, it must be set to the null pointer or null bytes.

For the MQPUT1 call, this field must be the null pointer or null bytes. This is because the response information (if requested) is returned in the response records specified by the object descriptor MQOD.

This is an input field. The initial value of this field is the null pointer in those programming languages that support pointers, and an all-null byte string otherwise. This field is not present if *Version* is less than MQPMO_VERSION_2.

Note: On platforms where the programming language does not support the pointer data type, this field is declared as a byte string of the appropriate length, with the initial value being the all-null byte string.

Table 46. Initial values of fields in MQPMO

Field name	Name of constant	Value of constant
StrucId	MQPMO_STRUC_ID	'PMOb' (See note 1)
Version	MQPMO_VERSION_1	1
Options	MQPMO_NONE	0
Timeout	None	-1
Context	None	0
KnownDestCount	None	0
UnknownDestCount	None	0
InvalidDestCount	None	0
ResolvedQName	None	Blanks (See note 2)
ResolvedQMgrName	None	Blanks
RecsPresent	None	0
PutMsgRecFields	MQPMRF_NONE	0
PutMsgRecOffset	None	0
ResponseRecOffset	None	0
PutMsgRecPtr	None	Null pointer or null bytes
ResponseRecPtr	None	Null pointer or null bytes

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQPMO_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQPMO MyPMO = {MQPMO_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQPMO {
  MQCHAR4   StrucId;           /* Structure identifier */
  MQLONG    Version;           /* Structure version number */
  MQLONG    Options;           /* Options that control the action of
                                  MQPUT or MQPUT1 */
  MQLONG    Timeout;           /* Reserved */
  MQHOBJ    Context;           /* Object handle of input queue */
  MQLONG    KnownDestCount;    /* Number of messages sent successfully
                                  to local queues */
```

```
  MQLONG    UnknownDestCount;   /* Number of messages sent successfully
                                   to remote queues */
  MQLONG    InvalidDestCount;   /* Number of messages that could not be
                                   sent */
  MQCHAR48  ResolvedQName;      /* Resolved name of destination queue */
  MQCHAR48  ResolvedQMgrName;   /* Resolved name of destination queue
                                   manager */
  MQLONG    RecsPresent;        /* Number of put message records or
                                   response records present */
  MQLONG    PutMsgRecFields;    /* Flags indicating which MQPMR fields
                                   are present */
  MQLONG    PutMsgRecOffset;    /* Offset of first put message record
                                   from start of MQPMO */
  MQLONG    ResponseRecOffset;  /* Offset of first response record from
                                   start of MQPMO */
  MQPTR     PutMsgRecPtr;       /* Address of first put message
                                   record */
  MQPTR     ResponseRecPtr;     /* Address of first response record */
 } MQPMO;
```

COBOL language declaration

```
**   MQPMO structure
  10 MQPMO.
**    Structure identifier
   15 MQPMO-STRUCID           PIC X(4).
**    Structure version number
   15 MQPMO-VERSION           PIC S9(9) BINARY.
**    Options that control the action of MQPUT or MQPUT1
   15 MQPMO-OPTIONS           PIC S9(9) BINARY.
**    Reserved
   15 MQPMO-TIMEOUT           PIC S9(9) BINARY.
**    Object handle of input queue
   15 MQPMO-CONTEXT           PIC S9(9) BINARY.
**    Number of messages sent successfully to local queues
   15 MQPMO-KNOWNDESTCOUNT    PIC S9(9) BINARY.
**    Number of messages sent successfully to remote queues
   15 MQPMO-UNKNOWNDESTCOUNT  PIC S9(9) BINARY.
**    Number of messages that could not be sent
   15 MQPMO-INVALIDDESTCOUNT  PIC S9(9) BINARY.
**    Resolved name of destination queue
   15 MQPMO-RESOLVEDQNAME     PIC X(48).
**    Resolved name of destination queue manager
   15 MQPMO-RESOLVEDQMGRNAME  PIC X(48).
**    Number of put message records or response records present
   15 MQPMO-RECSPRESENT       PIC S9(9) BINARY.
**    Flags indicating which MQPMR fields are present
   15 MQPMO-PUTMSGRECFIELDS   PIC S9(9) BINARY.
**    Offset of first put message record from start of MQPMO
   15 MQPMO-PUTMSGRECOFFSET   PIC S9(9) BINARY.
**    Offset of first response record from start of MQPMO
   15 MQPMO-RESPONSERECOFFSET PIC S9(9) BINARY.
**    Address of first put message record
   15 MQPMO-PUTMSGRECPTR      POINTER.
**    Address of first response record
   15 MQPMO-RESPONSERECPTR    POINTER.
```

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

```
dcl
 1 MQPMO based,
  3 StrucId           char(4),       /* Structure identifier */
  3 Version           fixed bin(31), /* Structure version number */
  3 Options           fixed bin(31), /* Options that control the action
                                        of MQPUT or MQPUT1 */
  3 Timeout           fixed bin(31), /* Reserved */
  3 Context           fixed bin(31), /* Object handle of input queue */
  3 KnownDestCount    fixed bin(31), /* Number of messages sent success-
                                        fully to local queues */
  3 UnknownDestCount  fixed bin(31), /* Number of messages sent success-
                                        fully to remote queues */
  3 InvalidDestCount  fixed bin(31), /* Number of messages that could
                                        not be sent */
  3 ResolvedQName     char(48),      /* Resolved name of destination
                                        queue */
  3 ResolvedQMgrName  char(48),      /* Resolved name of destination
                                        queue manager */
  3 RecsPresent       fixed bin(31), /* Number of put message records or
                                        response records present */
  3 PutMsgRecFields   fixed bin(31), /* Flags indicating which MQPMR
                                        fields are present */
  3 PutMsgRecOffset   fixed bin(31), /* Offset of first put message
                                        record from start of MQPMO */
  3 ResponseRecOffset fixed bin(31), /* Offset of first response record
                                        from start of MQPMO */
  3 PutMsgRecPtr      pointer,       /* Address of first put message
                                        record */
  3 ResponseRecPtr    pointer;       /* Address of first response
                                        record */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQPMO                   DSECT
MQPMO_STRUCID           DS   CL4      Structure identifier
MQPMO_VERSION           DS   F        Structure version number
MQPMO_OPTIONS           DS   F        Options that control the
*                                     action of MQPUT or MQPUT1
MQPMO_TIMEOUT           DS   F        Reserved
MQPMO_CONTEXT           DS   F        Object handle of input queue
MQPMO_KNOWNDESTCOUNT    DS   F        Reserved
MQPMO_UNKNOWNDESTCOUNT  DS   F        Reserved
MQPMO_INVALIDDESTCOUNT  DS   F        Reserved
MQPMO_RESOLVEDQNAME     DS   CL48     Resolved name of destination
*                                     queue
MQPMO_RESOLVEDQMGRNAME  DS   CL48     Resolved name of destination
*                                     queue manager
MQPMO_LENGTH            EQU  *-MQPMO  Length of structure
                        ORG  MQPMO
MQPMO_AREA              DS   CL(MQPMO_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQPMO^DEF (*);
BEGIN
STRUCT      STRUCID;
BEGIN STRING BYTE [0:3]; END;
INT(32)      VERSION;
INT(32)      OPTIONS;
INT(32)      TIMEOUT;
INT(32)      CONTEXT;
INT(32)      KNOWNDESTCOUNT;
INT(32)      UNKNOWNDESTCOUNT;
INT(32)      INVALIDDESTCOUNT;
STRUCT       RESOLVEDQNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT       RESOLVEDQMGRNAME;
BEGIN STRING BYTE [0:47]; END;
   END;
```

MQPMR – Put message record

The following table summarizes the fields in the structure.

Table 47. Fields in MQPMR

Field	Description	Page
MsgId	Message identifier	194
CorrelId	Correlation identifier	195
GroupId	Group identifier	195
Feedback	Feedback or reason code	195
AccountingToken	Accounting token	196

The MQPMR structure is used to specify various message properties for a single destination. By providing an array of these structures on the MQPUT or MQPUT1 call, it is possible to specify different values for each destination queue in a distribution list. Some of the fields are input only, others are input/output.

Note: This structure is unusual in that it does not have a fixed layout. The fields in this structure are optional, and the presence or absence of each field is indicated by the flags in the *PutMsgRecFields* field in MQPMO. Fields that are present ***must occur in the order shown below***. Fields that are absent occupy no space in the record.

Because MQPMR does not have a fixed layout, no declaration is provided for it in a header, COPY, and INCLUDE files for the supported programming languages. The application programmer should create a declaration containing the fields that are required by the application, and set the flags in *PutMsgRecFields* to indicate the fields that are present.

MQPMR is an input/output structure for the MQPUT and MQPUT1 calls.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields

MsgId (MQBYTE24)

Message identifier.

This is the message identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *MsgId* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *MsgId* field. If that value is MQMI_NONE, a new message identifier is generated for *each* of those destinations (that is, no two of those destinations have the same message identifier).

If MQPMO_NEW_MSG_ID is specified, new message identifiers are generated for all of the destinations in the distribution list, regardless of

whether they have MQPMR records. This is different from the way that MQPMO_NEW_CORREL_ID is processed (see below).

This is an input/output field.

CorrelId (MQBYTE24)
Correlation identifier.

This is the correlation identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *CorrelId* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *CorrelId* field.

If MQPMO_NEW_CORREL_ID is specified, a *single* new correlation identifier is generated and used for all of the destinations in the distribution list, regardless of whether they have MQPMR records. This is different from the way that MQPMO_NEW_MSG_ID is processed (see above).

This is an input/output field.

GroupId (MQBYTE24)
Group identifier.

This is the group identifier to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *GroupId* field in MQMD for a put to a single queue.

If this field is not present in the MQPMR record, or there are fewer MQPMR records than destinations, the value in MQMD is used for those destinations that do not have an MQPMR record containing a *GroupId* field. The value is processed as documented in Table 44 on page 179, but with the following differences:

- In those cases where a new group identifier would be used, the queue manager generates a different group identifier for each destination (that is, no two destinations have the same group identifier).
- In those cases where the value in the field would be used, the call fails with reason code MQRC_GROUP_ID_ERROR.

This is an input/output field.

Feedback (MQLONG)
Feedback or reason code.

This is the feedback code to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *Feedback* field in MQMD for a put to a single queue. If this field is not present, the value in MQMD is used.

This is an input field.

AccountingToken (MQBYTE32)
Accounting token.

This is the accounting token to be used for the message sent to the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call. It is processed in the same way as the *AccountingToken* field in MQMD for a put to a single queue. If this field is not present, the value in MQMD is used.

This is an input field.

There are no initial values defined for this structure, as no structure declarations are provided in the header, COPY, and INCLUDE files for the supported programming languages. The sample declarations below show how the structure should be declared by the application programmer if all of the fields are required.

C language declaration

```
typedef struct tagMQPMR {
  MQBYTE24  MsgId;            /* Message identifier */
  MQBYTE24  CorrelId;         /* Correlation identifier */
  MQBYTE24  GroupId;          /* Group identifier */
  MQLONG    Feedback;         /* Feedback or reason code */
  MQBYTE32  AccountingToken;  /* Accounting token */
 } MQPMR;
```

COBOL language declaration

```
**   MQPMR structure
  10 MQPMR.
**    Message identifier
   15 MQPMR-MSGID           PIC X(24).
**    Correlation identifier
   15 MQPMR-CORRELID        PIC X(24).
**    Group identifier
   15 MQPMR-GROUPID         PIC X(24).
**    Feedback or reason code
   15 MQPMR-FEEDBACK        PIC S9(9) BINARY.
**    Accounting token
   15 MQPMR-ACCOUNTINGTOKEN PIC X(32).
```

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

```
dcl
 1 MQPMR based,
  3 MsgId           char(24),      /* Message identifier */
  3 CorrelId        char(24),      /* Correlation identifier */
  3 GroupId         char(24),      /* Group identifier */
  3 Feedback        fixed bin(31), /* Feedback or reason code */
  3 AccountingToken char(32);      /* Accounting token */
```

MQRMH – Message reference header

The following table summarizes the fields in the structure.

Table 48. Fields in MQRMH

Field	Description	Page
StrucId	Structure identifier	198
Version	Structure version number	198
StrucLength	Total length of MQRMH, including strings at end of fixed fields, but not the bulk data	199
Encoding	Data encoding	199
CodedCharSetId	Coded character set identifier	199
Format	Format name	199
Flags	Reference message flags	199
ObjectType	Object type	200
ObjectInstanceId	Object instance identifier	200
SrcEnvLength	Length of source environment data	200
SrcEnvOffset	Offset of source environment data	200
SrcNameLength	Length of source object name	201
SrcNameOffset	Offset of source object name	201
DestEnvLength	Length of destination environment data	201
DestEnvOffset	Offset of destination environment data	201
DestNameLength	Length of destination object name	202
DestNameOffset	Offset of destination object name	202
DataLogicalLength	Length of bulk data	202
DataLogicalOffset	Low offset of bulk data	203
DataLogicalOffset2	High offset of bulk data	203

The MQRMH structure defines the format of a reference message header. An application can put a message in this format, omitting the bulk data. When the message is read from the transmission queue by a message channel agent (MCA), a user-supplied message exit is invoked to process the reference message header. The exit can append to the reference message the bulk data identified by the MQRMH structure, before the MCA sends the message through the channel to the next queue manager.

At the receiving end, a message exit that waits for reference messages should exist. When a reference message is received, the exit should create the object from the bulk data that follows the MQRMH in the message, and then pass on the reference message without the bulk data. The reference message can later be retrieved by an application reading the reference message (without the bulk data) from a queue.

Normally, the MQRMH structure (optionally with the bulk data) is all that is in the message. However, if the message is on a transmission queue, one or more additional headers will precede the MQRMH structure.

A reference message can also be sent to a distribution list. In this case, the MQDH structure and its related records precede the MQRMH structure when the message is on a transmission queue.

Note: A reference message should not be sent as a segmented message, because the message exit cannot process it correctly.

For data conversion purposes, conversion of the MQRMH structure includes conversion of the source environment data, source object name, destination environment data, and destination object name. Any other bytes within *StrucLength* are either discarded or have undefined values after data conversion. The bulk data will be converted provided that all of the following are true:

- The bulk data is present in the message when the data conversion is performed.
- The *Format* field in MQRMH has a value other than MQFMT_NONE.
- A user-written data-conversion exit exists with the format name specified.

Be aware, however, that usually the bulk data is *not* present in the message when the message is on a queue, and that as a result the bulk data will not be converted by the MQGMO_CONVERT option.

The format name of an MQRMH structure is MQFMT_REF_MSG_HEADER. The fields in the MQRMH structure, and the strings addressed by the offset fields, are in the character set and encoding given by the *CodedCharSetId* and *Encoding* fields in the header structure that precedes the MQRMH, or by those fields in the MQMD structure if the MQRMH is at the start of the application message data.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQRMH_STRUC_ID
Identifier for reference message header structure.

For the C programming language, the constant MQRMH_STRUC_ID_ARRAY is also defined; this has the same value as MQRMH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQRMH_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQRMH_VERSION_1
Version-1 reference message header structure.

The following constant specifies the version number of the current version:

MQRMH_CURRENT_VERSION
Current version of reference message header structure.

The initial value of this field is MQRMH_VERSION_1.

StrucLength (MQLONG)
Total length of MQRMH, including strings at end of fixed fields, but not the bulk data.

The initial value of this field is zero.

Encoding (MQLONG)
Data encoding.

This identifies the representation used for numeric values in the bulk data; this applies to binary integer data, packed-decimal integer data, and floating-point data.

The initial value of this field is MQENC_NATIVE.

CodedCharSetId (MQLONG)
Coded character set identifier.

This specifies the coded character set identifier of character data in the bulk data.

Note that character data in the MQ data structures must be in the character set used by the queue manager. This is defined by the queue manager's *CodedCharSetId* attribute; see "Attributes for the queue manager" on page 370 for details of this attribute.

The initial value of this field is `0`.

Format (MQCHAR8)
Format name.

This is a name that the sender of the message may use to indicate to the receiver the nature of the bulk data. 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 0 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.

The name should be padded with blanks to the length of the field. Do not use a null character to terminate the name before the end of the field, as the queue manager does not change the null and subsequent characters to blanks in the MQRMH structure. Do not specify a name with leading or embedded blanks.

The initial value of this field is MQFMT_NONE.

Flags (MQLONG)
Reference message flags.

The following flags are defined:

MQRMHF_LAST
Reference message contains or represents last part of object.

This flag indicates that the reference message represents or contains the last part of the referenced object.

MQRMHF_NOT_LAST
Reference message does not contain or represent last part of object.

MQRMHF_NOT_LAST is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

The initial value of this field is MQRMHF_NOT_LAST.

ObjectType (MQCHAR8)
Object type.

This is a name that can be used by the message exit to recognize types of reference message that it supports. It is recommended that the name conform to the same rules as the *Format* field described above.

The initial value of this field is 8 blanks.

ObjectInstanceId (MQBYTE24)
Object instance identifier.

This field can be used to identify a specific instance of an object. If it is not needed, it should be set to the following value:

MQOII_NONE
No object instance identifier specified.

The value is binary zero for the length of the field.

For the C programming language, the constant MQOII_NONE_ARRAY is also defined; this has the same value as MQOII_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_OBJECT_INSTANCE_ID_LENGTH. The initial value of this field is MQOII_NONE.

SrcEnvLength (MQLONG)
Length of source environment data.

If this field is zero, there is no source environment data, and *SrcEnvOffset* is ignored.

The initial value of this field is 0.

SrcEnvOffset (MQLONG)
Offset of source environment data.

This field specifies the offset of the source environment data from the start of the MQRMH structure. Source environment data can be specified by the creator of the reference message, if that data is known to the creator. For example, on OS/2 the source environment data might be the directory path of the object containing the bulk data. However, if the creator does not know the source environment data, it is the responsibility of the user-supplied message exit to determine any environment information needed.

The length of the source environment data is given by *SrcEnvLength*; if this length is zero, there is no source environment data, and *SrcEnvOffset* is ignored. If present, the source environment data must reside completely within *StrucLength* bytes from the start of the structure.

Applications should not assume that the environment data starts immediately after the last fixed field in the structure or that it is contiguous with any of the data addressed by the *SrcNameOffset*, *DestEnvOffset*, and *DestNameOffset* fields.

The initial value of this field is 0.

SrcNameLength (MQLONG)
Length of source object name.

If this field is zero, there is no source object name, and *SrcNameOffset* is ignored.

The initial value of this field is 0.

SrcNameOffset (MQLONG)
Offset of source object name.

This field specifies the offset of the source object name from the start of the MQRMH structure. The source object name can be specified by the creator of the reference message, if that data is known to the creator. However, if the creator does not know the source object name, it is the responsibility of the user-supplied message exit to identify the object to be accessed.

The length of the source object name is given by *SrcNameLength*; if this length is zero, there is no source object name, and *SrcNameOffset* is ignored. If present, the source object name must reside completely within *StrucLength* bytes from the start of the structure.

Applications should not assume that the source object name is contiguous with any of the data addressed by the *SrcEnvOffset*, *DestEnvOffset*, and *DestNameOffset* fields.

The initial value of this field is 0.

DestEnvLength (MQLONG)
Length of destination environment data.

If this field is zero, there is no destination environment data, and *DestEnvOffset* is ignored.

DestEnvOffset (MQLONG)
Offset of destination environment data.

This field specifies the offset of the destination environment data from the start of the MQRMH structure. Destination environment data can be specified by the creator of the reference message, if that data is known to the creator. For example, on OS/2 the destination environment data might be the directory path of the object where the bulk data is to be stored. However, if the creator does not know the destination environment data, it is the responsibility of the user-supplied message exit to determine any environment information needed.

The length of the destination environment data is given by *DestEnvLength*; if this length is zero, there is no destination environment data, and

DestEnvOffset is ignored. If present, the destination environment data must reside completely within *StrucLength* bytes from the start of the structure.

Applications should not assume that the destination environment data is contiguous with any of the data addressed by the *SrcEnvOffset*, *SrcNameOffset*, and *DestNameOffset* fields.

The initial value of this field is 0.

DestNameLength (MQLONG)
Length of destination object name.

If this field is zero, there is no destination object name, and *DestNameOffset* is ignored.

DestNameOffset (MQLONG)
Offset of destination object name.

This field specifies the offset of the destination object name from the start of the MQRMH structure. The destination object name can be specified by the creator of the reference message, if that data is known to the creator. However, if the creator does not know the destination object name, it is the responsibility of the user-supplied message exit to identify the object to be created or modified.

The length of the destination object name is given by *DestNameLength*; if this length is zero, there is no destination object name, and *DestNameOffset* is ignored. If present, the destination object name must reside completely within *StrucLength* bytes from the start of the structure.

Applications should not assume that the destination object name is contiguous with any of the data addressed by the *SrcEnvOffset*, *SrcNameOffset*, and *DestEnvOffset* fields.

The initial value of this field is 0.

DataLogicalLength (MQLONG)
Length of bulk data.

The *DataLogicalLength* field specifies the length of the bulk data referenced by the MQRMH structure.

If the bulk data is actually present in the message, the data begins at an offset of *StrucLength* bytes from the start of the MQRMH structure. The length of the entire message minus *StrucLength* gives the length of the bulk data present.

If data is present in the message, *DataLogicalLength* specifies the amount of that data that is relevant. The normal case is for *DataLogicalLength* to have the same value as the length of data actually present in the message.

If the MQRMH structure represents the remaining data in the object (starting from the specified logical offset), the value zero can be used for *DataLogicalLength*, provided that the bulk data is not actually present in the message.

If no data is present, the end of MQRMH coincides with the end of the message.

The initial value of this field is 0.

DataLogicalOffset (MQLONG)
Low offset of bulk data.

This field specifies the low offset of the bulk data from the start of the object of which the bulk data forms part. The offset of the bulk data from the start of the object is called the *logical offset*. This is *not* the physical offset of the bulk data from the start of the MQRMH structure – that offset is given by *StrucLength*.

To allow large objects to be sent using reference messages, the logical offset is divided into two fields, and the actual logical offset is given by the sum of these two fields:

- *DataLogicalOffset* represents the remainder obtained when the logical offset is divided by 1 000 000 000. It is thus a value in the range 0 through 999 999 999.
- *DataLogicalOffset2* represents the result obtained when the logical offset is divided by 1 000 000 000. It is thus the number of complete multiples of 1 000 000 000 that exist in the logical offset. The number of multiples is in the range 0 through 999 999 999.

The initial value of this field is 0.

DataLogicalOffset2 (MQLONG)
High offset of bulk data.

This field specifies the high offset of the bulk data from the start of the object of which the bulk data forms part. It is a value in the range 0 through 999 999 999. See *DataLogicalOffset* for details.

The initial value of this field is 0.

Table 49 (Page 1 of 2). Initial values of fields in MQRMH

Field name	Name of constant	Value of constant
StrucId	MQRMH_STRUC_ID	'RMHb' (See note 1)
Version	MQRMH_VERSION_1	1
StrucLength	None	0
Encoding	MQENC_NATIVE	See note 2
CodedCharSetId	None	0
Format	MQFMT_NONE	'bbbbbbbb'
Flags	MQRMHF_NOT_LAST	0
ObjectType	None	'bbbbbbbb'
ObjectInstanceId	MQOII_NONE	Nulls
SrcEnvLength	None	0
SrcEnvOffset	None	0
SrcNameLength	None	0
SrcNameOffset	None	0
DestEnvLength	None	0
DestEnvOffset	None	0
DestNameLength	None	0

Table 49 (Page 2 of 2). Initial values of fields in MQRMH

Field name	Name of constant	Value of constant
DestNameOffset	None	0
DataLogicalLength	None	0
DataLogicalOffset	None	0
DataLogicalOffset2	None	0

Notes:

1. The symbol 'b' represents a single blank character.
2. The value of this constant is environment-specific.
3. In the C programming language, the macro variable MQRMH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQRMH MyRMH = {MQRMH_DEFAULT};
   ```

C language declaration

```
typedef struct tagMQRMH {
  MQCHAR4   StrucId;             /* Structure identifier */
  MQLONG    Version;             /* Structure version number */
  MQLONG    StrucLength;         /* Total length of MQRMH, including
                                    strings at end of fixed fields, but
                                    not the bulk data */
  MQLONG    Encoding;            /* Data encoding */
  MQLONG    CodedCharSetId;      /* Coded character set identifier */
  MQCHAR8   Format;              /* Format name */
  MQLONG    Flags;               /* Reference message flags */
  MQCHAR8   ObjectType;          /* Object type */
  MQBYTE24  ObjectInstanceId;    /* Object instance identifier */
  MQLONG    SrcEnvLength;        /* Length of source environment data */
  MQLONG    SrcEnvOffset;        /* Offset of source environment data */
  MQLONG    SrcNameLength;       /* Length of source object name */
  MQLONG    SrcNameOffset;       /* Offset of source object name */
  MQLONG    DestEnvLength;       /* Length of destination environment
                                    data */
  MQLONG    DestEnvOffset;       /* Offset of destination environment
                                    data */
  MQLONG    DestNameLength;      /* Length of destination object name */
  MQLONG    DestNameOffset;      /* Offset of destination object name */
  MQLONG    DataLogicalLength;   /* Length of bulk data */
  MQLONG    DataLogicalOffset;   /* Low offset of bulk data */
  MQLONG    DataLogicalOffset2;  /* High offset of bulk data */
 } MQRMH;
```

COBOL language declaration

```
**   MQRMH structure
  10 MQRMH.
**    Structure identifier
   15 MQRMH-STRUCID            PIC X(4).
**    Structure version number
   15 MQRMH-VERSION            PIC S9(9) BINARY.
**    Total length of MQRMH, including strings at end of fixed
**    fields, but not the bulk data
```

```
   15 MQRMH-STRUCLENGTH        PIC S9(9) BINARY.
**    Data encoding
   15 MQRMH-ENCODING           PIC S9(9) BINARY.
**    Coded character set identifier
   15 MQRMH-CODEDCHARSETID     PIC S9(9) BINARY.
**    Format name
   15 MQRMH-FORMAT             PIC X(8).
**    Reference message flags
   15 MQRMH-FLAGS              PIC S9(9) BINARY.
**    Object type
   15 MQRMH-OBJECTTYPE         PIC X(8).
**    Object instance identifier
   15 MQRMH-OBJECTINSTANCEID   PIC X(24).
**    Length of source environment data
   15 MQRMH-SRCENVLENGTH       PIC S9(9) BINARY.
**    Offset of source environment data
   15 MQRMH-SRCENVOFFSET       PIC S9(9) BINARY.
**    Length of source object name
   15 MQRMH-SRCNAMELENGTH      PIC S9(9) BINARY.
**    Offset of source object name
   15 MQRMH-SRCNAMEOFFSET      PIC S9(9) BINARY.
**    Length of destination environment data
   15 MQRMH-DESTENVLENGTH      PIC S9(9) BINARY.
**    Offset of destination environment data
   15 MQRMH-DESTENVOFFSET      PIC S9(9) BINARY.
**    Length of destination object name
   15 MQRMH-DESTNAMELENGTH     PIC S9(9) BINARY.
**    Offset of destination object name
   15 MQRMH-DESTNAMEOFFSET     PIC S9(9) BINARY.
**    Length of bulk data
   15 MQRMH-DATALOGICALLENGTH  PIC S9(9) BINARY.
**    Low offset of bulk data
   15 MQRMH-DATALOGICALOFFSET  PIC S9(9) BINARY.
**    High offset of bulk data
   15 MQRMH-DATALOGICALOFFSET2 PIC S9(9) BINARY.
```

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

```
dcl
 1 MQRMH based,
  3 StrucId            char(4),       /* Structure identifier */
  3 Version            fixed bin(31), /* Structure version number */
  3 StrucLength        fixed bin(31), /* Total length of MQRMH,
                                         including strings at end of
                                         fixed fields, but not the bulk
                                         data */
  3 Encoding           fixed bin(31), /* Data encoding */
  3 CodedCharSetId     fixed bin(31), /* Coded character set
                                         identifier */
  3 Format             char(8),       /* Format name */
  3 Flags              fixed bin(31), /* Reference message flags */
  3 ObjectType         char(8),       /* Object type */
  3 ObjectInstanceId   char(24),      /* Object instance identifier */
  3 SrcEnvLength       fixed bin(31), /* Length of source environment
                                         data */
  3 SrcEnvOffset       fixed bin(31), /* Offset of source environment
                                         data */
  3 SrcNameLength      fixed bin(31), /* Length of source object name */
```

```
  3 SrcNameOffset      fixed bin(31), /* Offset of source object name */
  3 DestEnvLength      fixed bin(31), /* Length of destination environ-
                                         ment data */
  3 DestEnvOffset      fixed bin(31), /* Offset of destination environ-
                                         ment data */
  3 DestNameLength     fixed bin(31), /* Length of destination object
                                         name */
  3 DestNameOffset     fixed bin(31), /* Offset of destination object
                                         name */
  3 DataLogicalLength  fixed bin(31), /* Length of bulk data */
  3 DataLogicalOffset  fixed bin(31), /* Low offset of bulk data */
  3 DataLogicalOffset2 fixed bin(31); /* High offset of bulk data */
```

MQRR – Response record

The following table summarizes the fields in the structure.

Table 50. Fields in MQRR

Field	Description	Page
CompCode	Completion code for queue	207
Reason	Reason code for queue	207

The MQRR structure is used to receive the completion code and reason code resulting from the open or put operation for a single destination queue. By providing an array of these structures on the MQOPEN and MQPUT calls, or on the MQPUT1 call, it is possible to determine the completion codes and reason codes for all of the queues in a distribution list, when the outcome of the call is mixed, that is, when the call succeeds for some queues in the list, but fails for others. Reason code MQRC_MULTIPLE_REASONS from the call indicates that the response records (if provided by the application) have been set by the queue manager.

MQRR is an output structure for the MQOPEN, MQPUT, and MQPUT1 calls.

This structure is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Fields

CompCode (MQLONG)
Completion code for queue.

This is the completion code resulting from the open or put operation for the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call.

This is always an output field. The initial value of this field is MQCC_OK.

Reason (MQLONG)
Reason code for queue.

This is the reason code resulting from the open or put operation for the queue whose name was specified by the corresponding element in the array of MQOR structures provided on the MQOPEN or MQPUT1 call.

This is always an output field. The initial value of this field is MQRC_NONE.

Table 51. Initial values of fields in MQRR

Field name	Name of constant	Value of constant
CompCode	MQCC_OK	0
Reason	MQRC_NONE	0

Notes:

1. In the C programming language, the macro variable MQRR_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQRR MyRR = {MQRR_DEFAULT};
```

C language declaration

```
typedef struct tagMQRR {
  MQLONG  CompCode;  /* Completion code for queue */
  MQLONG  Reason;    /* Reason code for queue */
 } MQRR;
```

COBOL language declaration

```
**   MQRR structure
  10 MQRR.
**    Completion code for queue
   15 MQRR-COMPCODE PIC S9(9) BINARY.
**    Reason code for queue
   15 MQRR-REASON   PIC S9(9) BINARY.
```

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

```
dcl
 1 MQRR based,
  3 CompCode fixed bin(31), /* Completion code for queue */
  3 Reason   fixed bin(31); /* Reason code for queue */
```

MQTM – Trigger message

The following table summarizes the fields in the structure.

Table 52. Fields in MQTM

Field	Description	Page
StrucId	Structure identifier	210
Version	Structure version number	210
QName	Name of triggered queue	211
ProcessName	Name of process object	211
TriggerData	Trigger data	211
ApplType	Application type	212
ApplId	Application identifier	212
EnvData	Environment data	213
UserData	User data	213

The MQTM structure describes the data in the trigger message that is sent by the queue manager to a trigger-monitor application when a trigger event occurs for a queue. This structure is part of the MQSeries Trigger Monitor Interface (TMI), which is one of the MQSeries framework interfaces.

A trigger-monitor application may need to pass some or all of the information in the trigger message to the application which is started by the trigger-monitor application. Information which may be needed by the started application includes *QName*, *TriggerData*, and *UserData*. The trigger monitor application can pass the MQTM structure directly to the started application, or pass an MQTMC2 structure, depending on what is most convenient for the started application. For information about MQTMC2, see "MQTMC2 – Trigger message 2 (character format)" on page 217.

- On MVS/ESA, for an MQAT_CICS application that is started using the CKTI transaction, the entire trigger message structure MQTM is made available to the started transaction; the information can be retrieved by using the EXEC CICS RETRIEVE command.
- On OS/400, the trigger monitor application provided with MQSeries passes an MQTMC structure to the started application. The MQTMC structure is the same as MQTMC2, but with the *Version* field set to MQTMC_VERSION_1, and the *QMgrName* field omitted.
- On 16-bit Windows and 32-bit Windows, there is no trigger monitor application, and this structure is not supported.

For information about triggers, see the *MQSeries Application Programming Guide*.

The fields in the message descriptor of the trigger message are set as follows:

Field in MQMD	Value used
StrucId	MQMD_STRUC_ID
Version	MQMD_VERSION_1
Report	MQRO_NONE
MsgType	MQMT_DATAGRAM

Expiry	MQEI_UNLIMITED
Feedback	MQFB_NONE
Encoding	MQENC_NATIVE
CodedCharSetId	Queue manager's *CodedCharSetId* attribute
Format	MQFMT_TRIGGER
Priority	Initiation queue's *DefPriority* attribute
Persistence	MQPER_NOT_PERSISTENT
MsgId	A unique value
CorrelId	MQCI_NONE
BackoutCount	0
ReplyToQ	Blanks
ReplyToQMgr	Name of queue manager
UserIdentifier	Blanks
AccountingToken	MQACT_NONE
ApplIdentityData	Blanks
PutApplType	MQAT_QMGR, or as appropriate for the message channel agent
PutApplName	First 28 bytes of the queue-manager name
PutDate	Date when trigger message is sent
PutTime	Time when trigger message is sent
ApplOriginData	Blanks

An application that generates a trigger message is recommended to set similar values, except for the following:

- The *Priority* field can be set to MQPRI_PRIORITY_AS_Q_DEF (the queue manager will change this to the default priority for the initiation queue when the message is put).
- The *ReplyToQMgr* field can be set to blanks (the queue manager will change this to the name of the local queue manager when the message it put).
- The context fields should be set as appropriate for the application.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQTM_STRUC_ID
Identifier for trigger message structure.

For the C programming language, the constant MQTM_STRUC_ID_ARRAY is also defined; this has the same value as MQTM_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQTM_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQTM_VERSION_1
Version number for trigger message structure.

The following constant specifies the version number of the current version:

MQTM_CURRENT_VERSION
Current version of trigger message structure.

The initial value of this field is MQTM_VERSION_1.

QName (MQCHAR48)
Name of triggered queue.

This is the name of the queue for which a trigger event occurred, and is used by the application started by the trigger-monitor application. The queue manager initializes this field with the value of the *QName* attribute of the triggered queue; see “Attributes for all queues” on page 343 for details of this attribute.

Names that are shorter than the defined length of the field are padded to the right with blanks; they are not ended prematurely by a null character.

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

ProcessName (MQCHAR48)
Name of process object.

This is the name of the queue-manager process object specified for the triggered queue, and can be used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *ProcessName* attribute of the queue identified by the *QName* field; see “Attributes for local queues and model queues” on page 348 for details of this attribute.

Names that are shorter than the defined length of the field are always padded to the right with blanks; they are not ended prematurely by a null character.

The length of this field is given by MQ_PROCESS_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

TriggerData (MQCHAR64)
Trigger data.

This is free-format data for use by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *TriggerData* attribute of the queue identified by the *QName* field; see “Attributes for local queues and model queues” on page 348 for details of this attribute. The content of this data is of no significance to the queue manager.

On MVS/ESA, for a CICS application started using the CKTI transaction, this information is not used.

The length of this field is given by MQ_TRIGGER_DATA_LENGTH. The initial value of this field is the null string in C, and 64 blank characters in other programming languages.

ApplType (MQLONG)
Application type.

This identifies the nature of the program to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *ApplType* attribute of the process object identified by the *ProcessName* field; see "Attributes for process definitions" on page 367 for details of this attribute. The content of this data is of no significance to the queue manager.

ApplType can have one of the following standard values. User-defined types can also be used, but should be restricted to values in the range MQAT_USER_FIRST through MQAT_USER_LAST:

MQAT_AIX
AIX application (same value as MQAT_UNIX).
MQAT_CICS
CICS transaction.
MQAT_DOS
DOS client application.
MQAT_IMS
IMS application.
MQAT_MVS
MVS or TSO application.
MQAT_OS2
OS/2 or Presentation Manager application.
MQAT_OS400
OS/400 application.
MQAT_UNIX
UNIX application.
MQAT_WINDOWS
Windows client or 16-bit Windows application.
MQAT_WINDOWS_NT
Windows NT or 32-bit Windows application.
MQAT_USER_FIRST
Lowest value for user-defined application type.
MQAT_USER_LAST
Highest value for user-defined application type.

The initial value of this field is `0`.

ApplId (MQCHAR256)
Application identifier.

This is a character string that identifies the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *ApplId* attribute of the process object identified by the *ProcessName* field; see "Attributes for process definitions" on page 367 for details of this attribute. The content of this data is of no significance to the queue manager.

The interpretation to be placed on the information is determined by the trigger-monitor application. For example, *ApplId* could be:

- A program name (for MQAT_MVS applications)
- A CICS transaction ID (for MQAT_CICS applications).

On MVS/ESA, for a CICS application to be started using the CKTI transaction, or an IMS application to be started using the CSQQTRMN transaction, *ApplId* is a CICS or IMS transaction ID.

The length of this field is given by MQ_PROCESS_APPL_ID_LENGTH. The initial value of this field is the null string in C, and 256 blank characters in other programming languages.

EnvData (MQCHAR128)
Environment data.

This is a character string that contains environment-related information pertaining to the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *EnvData* attribute of the process object identified by the *ProcessName* field; see "Attributes for process definitions" on page 367 for details of this attribute. The content of this data is of no significance to the queue manager.

On MVS/ESA, for a CICS application started using the CKTI transaction, or an IMS application to be started using the CSQQTRMN transaction, this information is not used.

The length of this field is given by MQ_PROCESS_ENV_DATA_LENGTH. The initial value of this field is the null string in C, and 128 blank characters in other programming languages.

UserData (MQCHAR128)
User data.

This is a character string that contains user information relevant to the application to be started, and is used by the trigger-monitor application that receives the trigger message. The queue manager initializes this field with the value of the *UserData* attribute of the process object identified by the *ProcessName* field; see "Attributes for process definitions" on page 367 for details of this attribute. The content of this data is of no significance to the queue manager.

The length of this field is given by MQ_PROCESS_USER_DATA_LENGTH. The initial value of this field is the null string in C, and 128 blank characters in other programming languages.

Table 53. Initial values of fields in MQTM

Field name	Name of constant	Value of constant
StrucId	MQTM_STRUC_ID	'TMbb' (See note 1)
Version	MQTM_VERSION_1	1
QName	None	Blanks (See note 2)
ProcessName	None	Blanks
TriggerData	None	Blanks
ApplType	None	0
ApplId	None	Blanks
EnvData	None	Blanks
UserData	None	Blanks

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQTM_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

   ```
   MQTM MyTM = {MQTM_DEFAULT};
   ```

C language declaration

```
typedef 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 */
 } MQTM;
```

COBOL language declaration

```
**   MQTM structure
  10 MQTM.
**    Structure identifier
   15 MQTM-STRUCID     PIC X(4).
**    Structure version number
   15 MQTM-VERSION     PIC S9(9) BINARY.
**    Name of triggered queue
   15 MQTM-QNAME       PIC X(48).
**    Name of process object
   15 MQTM-PROCESSNAME PIC X(48).
**    Trigger data
   15 MQTM-TRIGGERDATA PIC X(64).
**    Application type
```

```
   15 MQTM-APPLTYPE    PIC S9(9) BINARY.
**    Application identifier
   15 MQTM-APPLID      PIC X(256).
**    Environment data
   15 MQTM-ENVDATA     PIC X(128).
**    User data
   15 MQTM-USERDATA    PIC X(128).
```

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

```
dcl
 1 MQTM based,
  3 StrucId     char(4),       /* Structure identifier */
  3 Version     fixed bin(31), /* Structure version number */
  3 QName       char(48),      /* Name of triggered queue */
  3 ProcessName char(48),      /* Name of process object */
  3 TriggerData char(64),      /* Trigger data */
  3 ApplType    fixed bin(31), /* Application type */
  3 ApplId      char(256),     /* Application identifier */
  3 EnvData     char(128),     /* Environment data */
  3 UserData    char(128);     /* User data */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQTM                            DSECT
MQTM_STRUCID                    DS   CL4      Structure identifier
MQTM_VERSION                    DS   F        Structure version number
MQTM_QNAME                      DS   CL48     Name of triggered queue
MQTM_PROCESSNAME                DS   CL48     Name of process object
MQTM_TRIGGERDATA                DS   CL64     Trigger data
MQTM_APPLTYPE                   DS   F        Application type
MQTM_APPLID                     DS   CL256    Application identifier
MQTM_ENVDATA                    DS   CL128    Environment data
MQTM_USERDATA                   DS   CL128    User data
MQTM_LENGTH                     EQU  *-MQTM   Length of structure
                                ORG  MQTM
MQTM_AREA                       DS   CL(MQTM_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQTM^DEF (*);
BEGIN
STRUCT       STRUCID;
BEGIN STRING BYTE [0:3]; END;
INT(32)      VERSION;
STRUCT       QNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT       PROCESSNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT       TRIGGERDATA;
BEGIN STRING BYTE [0:63]; END;
INT(32)      APPLTYPE;
  STRUCT       APPLID;
BEGIN STRING BYTE [0:255]; END;
STRUCT       ENVDATA;
BEGIN STRING BYTE [0:127]; END;
STRUCT       USERDATA;
BEGIN STRING BYTE [0:127]; END;
   END;
```

MQTMC2 – Trigger message 2 (character format)

The following table summarizes the fields in the structure.

Table 54. Fields in MQTMC2

Field	Description	Page
StrucId	Structure identifier	218
Version	Structure version number	218
QName	Name of triggered queue	218
ProcessName	Name of process object	218
TriggerData	Trigger data	218
ApplType	Application type	218
ApplId	Application identifier	218
EnvData	Environment data	218
UserData	User data	219
QMgrName	Queue manager name	219

When a trigger-monitor application retrieves a trigger message (MQTM) from an initiation queue, the trigger monitor may need to pass some or all of the information in the trigger message to the application that is started by the trigger monitor. Information that may be needed by the started application includes *QName*, *TriggerData*, and *UserData*. The trigger monitor application can pass the MQTM structure directly to the started application, or an MQTMC2 structure, depending on what is most convenient for the started application.

This structure is part of the MQSeries Trigger Monitor Interface (TMI), which is one of the MQSeries framework interfaces.

- On MVS/ESA, for an MQAT_IMS application that is started using the CSQQTRMN application, an MQTMC2 structure is made available to the started application.
- On OS/400, the trigger monitor application provided with MQSeries passes an MQTMC structure to the started application. The MQTMC structure is the same as MQTMC2, but with the *QMgrName* field omitted, and the *Version* field set to MQTMC_VERSION_1.
- On 16-bit Windows and 32-bit Windows, there is no trigger monitor application, and this structure is not supported.

The MQTMC2 structure is very similar to the format of the trigger message (MQTM structure). The difference is that the non-character fields in MQTM are changed in MQTMC2 to character fields of the same length, and the queue manager name is added at the end of the structure.

See “MQTM – Trigger message” on page 209 for details of the fields that are the same in this structure.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQTMC_STRUC_ID
Identifier for trigger message (character format) structure.

For the C programming language, the constant MQTMC_STRUC_ID_ARRAY is also defined; this has the same value as MQTMC_STRUC_ID, but is an array of characters instead of a string.

Version (MQCHAR4)
Structure version number.

The value must be:

MQTMC_VERSION_2
Version 2 trigger message (character format) structure.

For the C programming language, the constant MQTMC_VERSION_2_ARRAY is also defined; this has the same value as MQTMC_VERSION_2, but is an array of characters instead of a string.

The following constant specifies the version number of the current version:

MQTMC_CURRENT_VERSION
Current version of trigger message (character format) structure.

QName (MQCHAR48)
Name of triggered queue.

See the *QName* field in the MQTM structure.

ProcessName (MQCHAR48)
Name of process object.

See the *ProcessName* field in the MQTM structure.

TriggerData (MQCHAR64)
Trigger data.

See the *TriggerData* field in the MQTM structure.

ApplType (MQCHAR4)
Application type.

This field always contains blanks, whatever the value in the *ApplType* field in the MQTM structure of the original trigger message.

ApplId (MQCHAR256)
Application identifier.

See the *ApplId* field in the MQTM structure.

EnvData (MQCHAR128)
Environment data.

See the *EnvData* field in the MQTM structure.

UserData (MQCHAR128)
User data.

See the *UserData* field in the MQTM structure.

QMgrName (MQCHAR48)
Queue manager name.

This is the name of the queue manager at which the trigger event occurred.

Table 55. Initial values of fields in MQTMC2

Field name	Name of constant	Value of constant
StrucId	MQTMC_STRUC_ID	'TMCb' (See note 1)
Version	MQTMC_VERSION_2	'bbb2'
QName	None	Blanks (See note 2)
ProcessName	None	Blanks
TriggerData	None	Blanks
ApplType	None	'bbbb'
ApplId	None	Blanks
EnvData	None	Blanks
UserData	None	Blanks
QMgrName	None	Blanks

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQTMC2_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQTMC2 MyTMC = {MQTMC2_DEFAULT};
```

C language declaration

```
typedef struct tagMQTMC2 {
  MQCHAR4    StrucId;      /* Structure identifier */
  MQCHAR4    Version;      /* Structure version number */
  MQCHAR48   QName;        /* Name of triggered queue */
  MQCHAR48   ProcessName;  /* Name of process object */
  MQCHAR64   TriggerData;  /* Trigger data */
  MQCHAR4    ApplType;     /* Application type */
  MQCHAR256  ApplId;       /* Application identifier */
  MQCHAR128  EnvData;      /* Environment data */
  MQCHAR128  UserData;     /* User data */
  MQCHAR48   QMgrName;     /* Queue manager name */
 } MQTMC2;
```

COBOL language declaration

```
**   MQTMC2 structure
  10 MQTMC.
**    Structure identifier
   15 MQTMC-STRUCID     PIC X(4).
**    Structure version number
   15 MQTMC-VERSION     PIC X(4).
**    Name of triggered queue
   15 MQTMC-QNAME       PIC X(48).
**    Name of process object
   15 MQTMC-PROCESSNAME PIC X(48).
**    Trigger data
   15 MQTMC-TRIGGERDATA PIC X(64).
**    Application type
   15 MQTMC-APPLTYPE    PIC X(4).
**    Application identifier
   15 MQTMC-APPLID      PIC X(256).
**    Environment data
   15 MQTMC-ENVDATA     PIC X(128).
**    User data
   15 MQTMC-USERDATA    PIC X(128).
**    Queue manager name
   15 MQTMC-QMGRNAME    PIC X(48).
```

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

```
dcl
 1 MQTMC2 based,
  3 StrucId     char(4),   /* Structure identifier */
  3 Version     char(4),   /* Structure version number */
  3 QName       char(48),  /* Name of triggered queue */
  3 ProcessName char(48),  /* Name of process object */
  3 TriggerData char(64),  /* Trigger data */
  3 ApplType    char(4),   /* Application type */
  3 ApplId      char(256), /* Application identifier */
  3 EnvData     char(128), /* Environment data */
  3 UserData    char(128), /* User data */
  3 QMgrName    char(48);  /* Queue manager name */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQTMC                    DSECT
MQTMC2_STRUCID            DS   CL4      Structure identifier
MQTMC2_VERSION            DS   CL4      Structure version number
MQTMC2_QNAME              DS   CL48     Name of triggered queue
MQTMC2_PROCESSNAME        DS   CL48     Name of process object
MQTMC2_TRIGGERDATA        DS   CL64     Trigger data
MQTMC2_APPLTYPE           DS   CL4      Application type
MQTMC2_APPLID             DS   CL256    Application identifier
MQTMC2_ENVDATA            DS   CL128    Environment data
MQTMC2_USERDATA           DS   CL128    User data
MQTMC2_QMGRNAME           DS   CL48     Queue manager name
MQTMC2_LENGTH             EQU  *-MQTMC2 Length of structure
                          ORG  MQTMC
MQTMC2_AREA               DS   CL(MQTMC2_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQTMC2^DEF (*);
BEGIN
STRUCT     STRUCID;
BEGIN STRING BYTE [0:3]; END;
STRUCT     VERSION;
BEGIN STRING BYTE [0:3]; END;
STRUCT     QNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT       PROCESSNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT       TRIGGERDATA;
BEGIN STRING BYTE [0:63]; END;
STRUCT       APPLTYPE;
BEGIN STRING BYTE [0:3]; END;
STRUCT       APPLID;
BEGIN STRING BYTE [0:255]; END;
STRUCT       ENVDATA;
BEGIN STRING BYTE [0:127]; END;
STRUCT       USERDATA;
BEGIN STRING BYTE [0:127]; END;
STRUCT       QMQRNAME;
BEGIN STRING BYTE [0:47]; END;
END;
```

MQXP – Exit parameter block (MVS/ESA only)

The following table summarizes the fields in the structure.

Table 56. Fields in MQXP

Field	Description	Page
StrucId	Structure identifier	222
Version	Structure version number	222
ExitId	Exit identifier	223
ExitReason	Reason for invocation of exit	223
ExitResponse	Response from exit	223
ExitCommand	API call code	224
ExitParmCount	Parameter count	224
ExitUserArea	User area	224

The MQXP structure is used as an input/output variable to the API crossing exit. For more information on this exit, see the *MQSeries Application Programming Guide*.

This structure is supported only on MVS/ESA.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQXP_STRUC_ID
Identifier for exit parameter structure.

For the C programming language, the constant MQXP_STRUC_ID_ARRAY is also defined; this has the same value as MQXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)
Structure version number.

The value must be:

MQXP_VERSION_1
Version number for exit parameter-block structure.

Note: When a new version of this structure is introduced, the layout of the existing part is not changed. The exit should therefore check that the version number is equal to or greater than the lowest version that contains the fields that the exit needs to use.

This is an input field to the exit.

ExitId (MQLONG)
Exit identifier.

This is set on entry to the exit routine, and indicates the type of exit:

MQXT_API_CROSSING_EXIT
API crossing exit.

This is an input field to the exit.

On MVS/ESA and MQSeries for Windows, the channel message-retry exit and channel auto-definition exit are not supported.

ExitReason (MQLONG)
Reason for invocation of exit.

This is set on entry to the exit routine. For the API crossing exit it indicates whether the routine is called before or after execution of the API call:

MQXR_BEFORE
Before API execution.

MQXR_AFTER
After API execution.

ExitResponse (MQLONG)
Response from exit.

The value is set by the exit to communicate with the caller.

The following values are defined:

MQXCC_OK
Continue normally.

MQXCC_SUPPRESS_FUNCTION
Suppress function.

When this value is set by an API crossing exit called *before* the API call, the API call is not performed. The *CompCode* for the call is set to MQCC_OK, the *Reason* is set to MQRC_SUPPRESSED_BY_EXIT, and all other parameters remain as the exit left them.

When this value is set by an API crossing exit called *after* the API call, it is ignored by the queue manager.

MQXCC_SKIP_FUNCTION
Skip function.

When this value is set by an API crossing exit called *before* the API call, the API call is not performed; the *CompCode* and *Reason* and all other parameters remain as the exit left them.

When this value is set by an API crossing exit called *after* the API call, it is ignored by the queue manager.

This is an output field from the exit.

ExitCommand (MQLONG)
API call code.

This field is set on entry to the exit routine. It identifies the API call that caused the exit to be invoked:

MQXC_MQBACK
The MQBACK call.

MQXC_MQCLOSE
The MQCLOSE call.

MQXC_MQCMIT
The MQCMIT call.

MQXC_MQGET
The MQGET call.

MQXC_MQINQ
The MQINQ call.

MQXC_MQOPEN
The MQOPEN call.

MQXC_MQPUT
The MQPUT call.

MQXC_MQPUT1
The MQPUT1 call.

MQXC_MQSET
The MQSET call.

This is an input field to the exit.

ExitParmCount (MQLONG)
Parameter count.

This field is set on entry to the exit routine. It contains the number of parameters that the API call takes. These are:

MQBACK	3
MQCLOSE	5
MQCMIT	3
MQGET	9
MQINQ	10
MQOPEN	6
MQPUT	8
MQPUT1	8
MQSET	10

This is an input field to the exit.

Reserved (MQLONG)
Reserved.

This is a reserved field. Its value is not significant to the exit.

ExitUserArea (MQBYTE16)
User area.

This is a field that is available for the exit to use. It is initialized to binary zero for the length of the field before the first invocation of the exit for the

task, and thereafter any changes made to this field by the exit are preserved across invocations of the exit.

The following value is defined:

MQXUA_NONE
No user information.

The value is binary zero for the length of the field.

For the C programming language, the constant MQXUA_NONE_ARRAY is also defined; this has the same value as MQXUA_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_EXIT_USER_AREA_LENGTH.

This is an input/output field to the exit.

C language declaration

```
typedef struct tagMQXP {
  MQCHAR4   StrucId;        /* Structure identifier */
  MQLONG    Version;        /* Structure version number */
  MQLONG    ExitId;         /* Exit identifier */
  MQLONG    ExitReason;     /* Reason for invocation of exit */
  MQLONG    ExitResponse;   /* Response from exit */
  MQLONG    ExitCommand;    /* API call code */
  MQLONG    ExitParmCount;  /* Parameter count */
  MQLONG    Reserved;       /* Reserved */
  MQBYTE16  ExitUserArea;   /* User area */
 } MQXP;
```

COBOL language declaration

```
**   MQXP structure
  10 MQXP.
**    Structure identifier
   15 MQXP-STRUCID       PIC X(4).
**    Structure version number
   15 MQXP-VERSION       PIC S9(9) BINARY.
**    Exit identifier
   15 MQXP-EXITID        PIC S9(9) BINARY.
**    Reason for invocation of exit
   15 MQXP-EXITREASON    PIC S9(9) BINARY.
**    Response from exit
   15 MQXP-EXITRESPONSE  PIC S9(9) BINARY.
**    API call code
   15 MQXP-EXITCOMMAND   PIC S9(9) BINARY.
**    Parameter count
   15 MQXP-EXITPARMCOUNT PIC S9(9) BINARY.
**    Reserved
   15 MQXP-RESERVED      PIC S9(9) BINARY.
**    User area
   15 MQXP-EXITUSERAREA  PIC X(16).
```

PL/I language declaration

```
dcl
 1 MQXP based,
  3 StrucId       char(4),       /* Structure identifier */
  3 Version       fixed bin(31), /* Structure version number */
  3 ExitId        fixed bin(31), /* Exit identifier */
  3 ExitReason    fixed bin(31), /* Reason for invocation of exit */
  3 ExitResponse  fixed bin(31), /* Response from exit */
  3 ExitCommand   fixed bin(31), /* API call code */
  3 ExitParmCount fixed bin(31), /* Parameter count */
  3 Reserved      fixed bin(31), /* Reserved */
  3 ExitUserArea  char(16);      /* User area */
```

System/390 assembler language declaration

```
MQXP                    DSECT
MQXP_STRUCID            DS   CL4      Structure identifier
MQXP_VERSION            DS   F        Structure version number
MQXP_EXITID             DS   F        Exit identifier
MQXP_EXITREASON         DS   F        Reason for invocation of
*                                     exit
MQXP_EXITRESPONSE       DS   F        Response from exit
MQXP_EXITCOMMAND        DS   F        API call code
MQXP_EXITPARMCOUNT      DS   F        Parameter count
MQXP_RESERVED           DS   F        Reserved
MQXP_EXITUSERAREA       DS   XL16     User area
MQXP_LENGTH             EQU  *-MQXP   Length of structure
                        ORG  MQXP
MQXP_AREA               DS   CL(MQXP_LENGTH)
```

MQXQH – Transmission queue header

The following table summarizes the fields in the structure.

Table 57. Fields in MQXQH

Field	Description	Page
StrucId	Structure identifier	230
Version	Structure version number	230
RemoteQName	Name of destination queue	230
RemoteQMgrName	Name of destination queue manager	230
MsgDesc	Original message descriptor	231

The MQXQH structure describes the information that is prefixed to the application message data of messages when they are on transmission queues. A transmission queue is a special type of local queue that temporarily holds messages destined for remote queues (that is, destined for queues that do not belong to the local queue manager). A transmission queue is denoted by the *Usage* queue attribute having the value MQUS_TRANSMISSION.

A message that is on a transmission queue has *two* message descriptors:

- One message descriptor is stored separately from the message data; this is called the *separate message descriptor*, and is a modified version of the message descriptor provided by the application in the *MsgDesc* parameter of the MQPUT or MQPUT1 call (see below for details).

  The message put by the application may be a message in a group, or a segment of a logical message, or may have segmentation allowed, but these properties are *not* propagated into the separate message descriptor – the version-2 fields in the separate message descriptor always have their default values.

  The separate message descriptor is the one that is returned to the application in the *MsgDesc* parameter of the MQGET call when the message is removed from the transmission queue.

- A second message descriptor is stored within the MQXQH structure, as part of the message data; this is called the *embedded message descriptor*, and is a close copy of the message descriptor that was provided by the application in the *MsgDesc* parameter of the MQPUT or MQPUT1 call (see below for details).

  The embedded message descriptor is always a version-1 MQMD. If the message put by the application has nondefault values for one or more of the version-2 fields in the MQMD, an MQMDE structure follows the MQXQH, and is in turn followed by the application message data (if any). The MQMDE is either:

  – Generated by the queue manager (if the application uses a version-2 MQMD to put the message), or

  – Already present at the start of the application message data (if the application uses a version-1 MQMD to put the message).

  The embedded message descriptor is the one that is returned to the application in the *MsgDesc* parameter of the MQGET call when the message is removed from the final destination queue.

Putting messages on remote queues: When an application puts a message on a remote queue (either by specifying the name of the remote queue directly, or by using a local definition of the remote queue), the local queue manager:

- Creates an MQXQH structure containing the embedded message descriptor
- Appends an MQMDE if one is needed and is not already present
- Appends the application message data
- Places the message on an appropriate transmission queue

Character data in the MQXQH structure is in the character set of the local queue manager (defined by the *CodedCharSetId* queue manager attribute), and integer data is in the native machine encoding. These values are stored in the separate message descriptor, and may be different from the values of the *CodedCharSetId* and *Encoding* fields in the embedded message descriptor, because the latter fields relate to the application message data and not the MQXQH structure itself.

The fields in the embedded message descriptor have the same values as those in the *MsgDesc* parameter of the MQPUT or MQPUT1 call, with the exception of the following:

- The *Version* field always has the value MQMD_VERSION_1.
- If the *Priority* field has the value MQPRI_PRIORITY_AS_Q_DEF, it is replaced by the value of the queue's *DefPriority* attribute.
- If the *Persistence* field has the value MQPER_PERSISTENCE_AS_Q_DEF, it is replaced by the value of the queue's *DefPersistence* attribute.
- If the *MsgId* field has the value MQMI_NONE, or the MQPMO_NEW_MSG_ID option was specified, or the message is a distribution-list message, *MsgId* is replaced by a new message identifier generated by the queue manager.

  When a distribution-list message is split into smaller distribution-list messages placed on different transmission queues, the *MsgId* field in each of the new embedded message descriptors is the same as that in the original distribution-list message.
- If the MQPMO_NEW_CORREL_ID option was specified, *CorrelId* is replaced by a new correlation identifier generated by the queue manager.
- The context fields are set as indicated by the MQPMO_*_CONTEXT context option(s) specified in the *PutMsgOpts* parameter; the context fields are the fields *UserIdentifier* through *ApplOriginData* in the list below.
- The version-2 fields (if they were present) are removed from the MQMD, and moved into an MQMDE structure, if one or more of the version-2 fields has a nondefault value.

The fields in the separate message descriptor are set by the queue manager as shown below. If the queue manager does not support the version-2 MQMD, a version-1 MQMD is used without loss of function.

Field in separate MQMD	Value used
StrucId	MQMD_STRUC_ID
Version	MQMD_VERSION_2
Report	Copied from the embedded message descriptor, but with the bits identified by MQRO_ACCEPT_UNSUP_IF_XMIT_MASK set to zero. (This prevents a COA or COD report message being generated when a message is placed on or removed from a transmission queue.)

Field in separate MQMD	Value used
MsgType	Copied from the embedded message descriptor.
Expiry	Copied from the embedded message descriptor.
Feedback	Copied from the embedded message descriptor.
Encoding	MQENC_NATIVE (see note below)
CodedCharSetId	Queue manager's *CodedCharSetId* attribute.
Format	MQFMT_XMIT_Q_HEADER
Priority	Copied from the embedded message descriptor.
Persistence	Copied from the embedded message descriptor.
MsgId	A new value is generated by the queue manager. This message identifier is different from the *MsgId* that the queue manager may have generated for the embedded message descriptor (see above).
CorrelId	The *MsgId* from the embedded message descriptor.
BackoutCount	0
ReplyToQ	Copied from the embedded message descriptor.
ReplyToQMgr	Copied from the embedded message descriptor.
UserIdentifier	Copied from the embedded message descriptor.
AccountingToken	Copied from the embedded message descriptor.
ApplIdentityData	Copied from the embedded message descriptor.
PutApplType	MQAT_QMGR
PutApplName	First 28 bytes of the queue-manager name.
PutDate	Date when message was put on transmission queue.
PutTime	Time when message was put on transmission queue.
ApplOriginData	Blanks
GroupId	MQGI_NONE
MsgSeqNumber	1
Offset	0
MsgFlags	MQMF_NONE
OriginalLength	MQOL_UNDEFINED

On OS/2 and Windows NT, the value of MQENC_NATIVE for Micro Focus COBOL differs from the value for C. The value in the *Encoding* field in the separate message descriptor is always the value for C in these environments; this value is 546 in decimal. Also, the integer fields in the MQXQH structure are in the encoding that corresponds to this value (the native Intel encoding).

Putting messages directly on transmission queues: It is also possible for an application to put a message directly on a transmission queue. In this case the application must prefix the application message data with an MQXQH structure, and initialize the fields with appropriate values. In addition, the *Format* field in the *MsgDesc* parameter of the MQPUT or MQPUT1 call must have the value MQFMT_XMIT_Q_HEADER.

Character data in the MQXQH structure created by the application must be in the character set of the local queue manager (defined by the *CodedCharSetId* queue-manager attribute), and integer data must be in the native machine encoding. In addition, character data in the MQXQH structure must be padded with blanks to the defined length of the field; the data must not be ended prematurely by using a null character, because the queue manager does not convert the null and subsequent characters to blanks in the MQXQH structure.

Note however that the queue manager does not check that an MQXQH structure is present, or that valid values have been specified for the fields.

Getting messages from transmission queues: Applications that get messages from a transmission queue must process the information in the MQXQH structure in

an appropriate fashion. The presence of the MQXQH structure at the beginning of the application message data is indicated by the value MQFMT_XMIT_Q_HEADER being returned in the *Format* field in the *MsgDesc* parameter of the MQGET call. The values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter indicate the character set and encoding of the character and integer data in the MQXQH structure, respectively. The character set and encoding of the application message data are defined by the *CodedCharSetId* and *Encoding* fields in the embedded message descriptor.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQXQH_STRUC_ID
Identifier for transmission-queue header structure.

For the C programming language, the constant MQXQH_STRUC_ID_ARRAY is also defined; this has the same value as MQXQH_STRUC_ID, but is an array of characters instead of a string.

The initial value of this field is MQXQH_STRUC_ID.

Version (MQLONG)
Structure version number.

The value must be:

MQXQH_VERSION_1
Version number for transmission-queue header structure.

The following constant specifies the version number of the current version:

MQXQH_CURRENT_VERSION
Current version of transmission-queue header structure.

The initial value of this field is MQXQH_VERSION_1.

RemoteQName (MQCHAR48)
Name of destination queue.

This is the name of the message queue that is the apparent eventual destination for the message (this may prove not to be the actual eventual destination if, for example, this queue is defined at *RemoteQMgrName* to be a local definition of another remote queue).

If the message is a distribution-list message (that is, the *Format* field in the embedded message descriptor is MQFMT_DIST_HEADER), *RemoteQName* is blank.

The length of this field is given by MQ_Q_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

RemoteQMgrName (MQCHAR48)
Name of destination queue manager.

This is the name of the queue manager that owns the queue that is the apparent eventual destination for the message.

If the message is a distribution-list message, *RemoteQMgrName* is blank.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. The initial value of this field is the null string in C, and 48 blank characters in other programming languages.

MsgDesc (MQMD1)
Original message descriptor.

This is the embedded message descriptor, and is a close copy of the message descriptor MQMD that was specified as the *MsgDesc* parameter on the MQPUT or MQPUT1 call when the message was originally put to the remote queue.

Note: This is a version-1 MQMD.

The initial values of the fields in this structure are the same as those in the MQMD structure.

Table 58. Initial values of fields in MQXQH

Field name	Name of constant	Value of constant
StrucId	MQXQH_STRUC_ID	'XQHb' (See note 1)
Version	MQXQH_VERSION_1	1
RemoteQName	None	Blanks (See note 2)
RemoteQMgrName	None	Blanks
MsgDesc	Same names and values as for MQMD; see Table 35 on page 147	

Notes:

1. The symbol 'b' represents a single blank character.
2. The value 'Blanks' denotes the null string in C, and blank characters in other programming languages.
3. In the C programming language, the macro variable MQXQH_DEFAULT contains the values listed above. It can be used in the following way to provide initial values for the fields in the structure:

```
MQXQH MyXQH = {MQXQH_DEFAULT};
```

C language declaration

```
typedef struct tagMQXQH {
  MQCHAR4   StrucId;         /* Structure identifier */
  MQLONG    Version;         /* Structure version number */
  MQCHAR48  RemoteQName;     /* Name of destination queue */
  MQCHAR48  RemoteQMgrName;  /* Name of destination queue manager */
  MQMD1     MsgDesc;         /* Original message descriptor */
 } MQXQH;
```

COBOL language declaration

```
**   MQXQH structure
  10 MQXQH.
**    Structure identifier
   15 MQXQH-STRUCID                  PIC X(4).
**    Structure version number
   15 MQXQH-VERSION                  PIC S9(9) BINARY.
**    Name of destination queue
   15 MQXQH-REMOTEQNAME              PIC X(48).
**    Name of destination queue manager
   15 MQXQH-REMOTEQMGRNAME           PIC X(48).
**    Original message descriptor
   15 MQXQH-MSGDESC.
**     Structure identifier
    20 MQXQH-MSGDESC-STRUCID          PIC X(4).
**     Structure version number
    20 MQXQH-MSGDESC-VERSION          PIC S9(9) BINARY.
**     Report options
    20 MQXQH-MSGDESC-REPORT           PIC S9(9) BINARY.
**     Message type
    20 MQXQH-MSGDESC-MSGTYPE          PIC S9(9) BINARY.
**     Expiry time
    20 MQXQH-MSGDESC-EXPIRY           PIC S9(9) BINARY.
**     Feedback or reason code
    20 MQXQH-MSGDESC-FEEDBACK         PIC S9(9) BINARY.
**     Data encoding
    20 MQXQH-MSGDESC-ENCODING         PIC S9(9) BINARY.
**     Coded character set identifier
    20 MQXQH-MSGDESC-CODEDCHARSETID   PIC S9(9) BINARY.
**     Format name
    20 MQXQH-MSGDESC-FORMAT           PIC X(8).
**     Message priority
    20 MQXQH-MSGDESC-PRIORITY         PIC S9(9) BINARY.
**     Message persistence
    20 MQXQH-MSGDESC-PERSISTENCE      PIC S9(9) BINARY.
**     Message identifier
    20 MQXQH-MSGDESC-MSGID            PIC X(24).
**     Correlation identifier
    20 MQXQH-MSGDESC-CORRELID         PIC X(24).
**     Backout counter
    20 MQXQH-MSGDESC-BACKOUTCOUNT     PIC S9(9) BINARY.
**     Name of reply-to queue
    20 MQXQH-MSGDESC-REPLYTOQ         PIC X(48).
**     Name of reply queue manager
    20 MQXQH-MSGDESC-REPLYTOQMGR      PIC X(48).
**     User identifier
    20 MQXQH-MSGDESC-USERIDENTIFIER   PIC X(12).
**     Accounting token
    20 MQXQH-MSGDESC-ACCOUNTINGTOKEN  PIC X(32).
**     Application data relating to identity
    20 MQXQH-MSGDESC-APPLIDENTITYDATA PIC X(32).
**     Type of application that put the message
    20 MQXQH-MSGDESC-PUTAPPLTYPE      PIC S9(9) BINARY.
**     Name of application that put the message
    20 MQXQH-MSGDESC-PUTAPPLNAME      PIC X(28).
**     Date when message was put
```

```
    20 MQXQH-MSGDESC-PUTDATE          PIC X(8).
**     Time when message was put
    20 MQXQH-MSGDESC-PUTTIME          PIC X(8).
**     Application data relating to origin
    20 MQXQH-MSGDESC-APPLORIGINDATA   PIC X(4).
```

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

```
dcl
 1 MQXQH based,
  3 StrucId           char(4),       /* Structure identifier */
  3 Version           fixed bin(31), /* Structure version number */
  3 RemoteQName       char(48),      /* Name of destination queue */
  3 RemoteQMgrName    char(48),      /* Name of destination queue
                                        manager */
  3 MsgDesc,                         /* Original message descriptor */
   5 StrucId          char(4),       /* Structure identifier */
   5 Version          fixed bin(31), /* Structure version number */
   5 Report           fixed bin(31), /* Report options */
   5 MsgType          fixed bin(31), /* Message type */
   5 Expiry           fixed bin(31), /* Expiry time */
   5 Feedback         fixed bin(31), /* Feedback or reason code */
   5 Encoding         fixed bin(31), /* Data encoding */
   5 CodedCharSetId   fixed bin(31), /* Coded character set
                                        identifier */
   5 Format           char(8),       /* Format name */
   5 Priority         fixed bin(31), /* Message priority */
   5 Persistence      fixed bin(31), /* Message persistence */
   5 MsgId            char(24),      /* Message identifier */
   5 CorrelId         char(24),      /* Correlation identifier */
   5 BackoutCount     fixed bin(31), /* Backout counter */
   5 ReplyToQ         char(48),      /* Name of reply-to queue */
   5 ReplyToQMgr      char(48),      /* Name of reply queue manager */
   5 UserIdentifier   char(12),      /* User identifier */
   5 AccountingToken  char(32),      /* Accounting token */
   5 ApplIdentityData char(32),      /* Application data relating to
                                        identity */
   5 PutApplType      fixed bin(31), /* Type of application that put the
                                        message */
   5 PutApplName      char(28),      /* Name of application that put the
                                        message */
   5 PutDate          char(8),       /* Date when message was put */
   5 PutTime          char(8),       /* Time when message was put */
   5 ApplOriginData   char(4);       /* Application data relating to
                                        origin */
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQXQH                           DSECT
MQXQH_STRUCID                   DS   CL4      Structure identifier
MQXQH_VERSION                   DS   F        Structure version number
MQXQH_REMOTEQNAME               DS   CL48     Name of destination queue
MQXQH_REMOTEQMGRNAME            DS   CL48     Name of destination queue
*                                            manager
MQXQH_MSGDESC                   DS   0F       Force fullword alignment
MQXQH_MSGDESC_STRUCID           DS   CL4      Structure identifier
MQXQH_MSGDESC_VERSION           DS   F        Structure version number
MQXQH_MSGDESC_REPORT            DS   F        Report options
MQXQH_MSGDESC_MSGTYPE           DS   F        Message type
MQXQH_MSGDESC_EXPIRY            DS   F        Expiry time
MQXQH_MSGDESC_FEEDBACK          DS   F        Feedback or reason code
MQXQH_MSGDESC_ENCODING          DS   F        Data encoding
MQXQH_MSGDESC_CODEDCHARSETID    DS   F        Coded character set
*                                            identifier
MQXQH_MSGDESC_FORMAT            DS   CL8      Format name
MQXQH_MSGDESC_PRIORITY          DS   F        Message priority
MQXQH_MSGDESC_PERSISTENCE       DS   F        Message persistence
MQXQH_MSGDESC_MSGID             DS   XL24     Message identifier
MQXQH_MSGDESC_CORRELID          DS   XL24     Correlation identifier
MQXQH_MSGDESC_BACKOUTCOUNT      DS   F        Backout counter
MQXQH_MSGDESC_REPLYTOQ          DS   CL48     Name of reply-to queue
MQXQH_MSGDESC_REPLYTOQMGR       DS   CL48     Name of reply queue manager
MQXQH_MSGDESC_USERIDENTIFIER    DS   CL12     User identifier
MQXQH_MSGDESC_ACCOUNTINGTOKEN   DS   XL32     Accounting token
MQXQH_MSGDESC_APPLIDENTITYDATA  DS   CL32     Application data relating to
*                                            identity
MQXQH_MSGDESC_PUTAPPLTYPE       DS   F        Type of application that put
*                                            the message
MQXQH_MSGDESC_PUTAPPLNAME       DS   CL28     Name of application that put
*                                            the message
MQXQH_MSGDESC_PUTDATE           DS   CL8      Date when message was put
MQXQH_MSGDESC_PUTTIME           DS   CL8      Time when message was put
MQXQH_MSGDESC_APPLORIGINDATA    DS   CL4      Application data relating to
*                                            origin
MQXQH_MSGDESC_LENGTH            EQU  *-MQXQH_MSGDESC
                                ORG  MQXQH_MSGDESC
MQXQH_MSGDESC_AREA              DS   CL(MQXQH_MSGDESC_LENGTH)
MQXQH_LENGTH                    EQU  *-MQXQH  Length of structure
                                ORG  MQXQH
MQXQH_AREA                      DS   CL(MQXQH_LENGTH)
```

TAL declaration (Tandem NSK only)

```
STRUCT       MQXQH^DEF (*);
BEGIN
STRUCT        STRUCID;
BEGIN STRING BYTE [0:3]; END;
INT(32)       VERSION;
STRUCT        REMOTEQNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT        REMOTEQMGRNAME;
BEGIN STRING BYTE [0:47]; END;
STRUCT      MSGDESC(MQMD^DEF);
END;
```

Chapter 3. Call descriptions

This chapter describes the MQI calls:

- MQBACK – Back out
- MQBEGIN – Begin unit of work
- MQCLOSE – Close object
- MQCMIT – Commit
- MQCONN – Connect to queue manager
- MQCONNX – Connect to queue manager with options
- MQDISC – Disconnect from queue manager
- MQGET – Get message
- MQINQ – Inquire about object attributes
- MQOPEN – Open object
- MQPUT – Put message
- MQPUT1 – Put one message
- MQSET – Set object attributes
- MQSYNC – Synchronize statistics updates (Tandem NSK only)

Online help on the UNIX platforms, in the form of *man* pages, is available for these calls.

Note: The calls associated with data conversion, MQXCNVC and MQDATACONVEXIT, are in Appendix D, “Data-conversion” on page 495.

Conventions used in the call descriptions

For each call, this chapter gives a description of the parameters and usage of the call in a format that is independent of programming language. This is followed by typical invocations of the call, and typical declarations of its parameters, in each of the supported programming languages.

The description of each call contains the following sections:

Call name The call name, followed by a brief description of the purpose of the call.

Parameters For each parameter, the name is followed by its data type in parentheses ( ) and one of the following:

input You supply information in the parameter when you make the call.

output The queue manager returns information in the parameter when the call completes or fails.

input/output You supply information in the parameter when you make the call, and the queue manager changes the information when the call completes or fails.

For example:

Compcode (MQLONG) — output

In some cases, the data type is a structure. In all cases, there is more information about the data type or structure in Chapter 1, “Data type descriptions – elementary” on page 1.

The last two parameters in each call are a completion code and a reason code. The completion code indicates whether the call completed successfully, partially, or not at all. Further information about the partial success or the failure of the call is given in the reason code. You will find more information about each completion and reason code in Chapter 5, “Return codes” on page 383.

Usage notes
Additional information about the call, describing how to use it and any restrictions on its use.

Assembler language invocation
Typical invocation of the call, and declaration of its parameters, in assembler language.

C invocation
Typical invocation of the call, and declaration of its parameters, in C.

COBOL invocation
Typical invocation of the call, and declaration of its parameters, in COBOL.

PL/I invocation
Typical invocation of the call, and declaration of its parameters, in PL/I.

All parameters are passed by reference.

Other notation conventions are:

Constants Names of constants are shown in uppercase; for example, MQOO_OUTPUT. A set of constants having the same prefix is shown like this: MQIA_★. See Chapter 6, “MQSeries constants” on page 449 for the value of a constant.

Arrays In some calls, parameters are arrays of character strings whose size is not fixed. In the descriptions of these parameters, a lowercase “n” represents a numeric constant. When you code the declaration for that parameter, replace the “n” with the numeric value you require.

Using the calls in the C language

Parameters that are *input only* and of 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 parameter data. Parameters for which this is possible are identified in the call descriptions.

No parameter is returned as the value of the call; in C terminology, this means that all calls return **void**.

Declaring the Buffer parameter

The **MQGET**, **MQPUT**, and **MQPUT1** calls each have one parameter that has an undefined data type—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 perfectly valid to declare the parameters in this way, but it is usually more convenient to declare them as the particular structure that describes the layout of the data in the message. The function prototype declares the parameter as a pointer-to-void, so that you can specify the address of any sort of data as the parameter on the call invocation.

Pointer-to-void is a pointer to data of undefined format. It is defined as:

```
typedef void *PMQVOID;
```

MQBACK – Back out changes

The MQBACK call indicates to the queue manager that all of the message gets and puts that have occurred since the last syncpoint are to be backed out. Messages put as part of a unit of work are deleted; messages retrieved as part of a unit of work are reinstated on the queue.

- On MVS/ESA, this call is used only by batch programs (including IMS batch DL/I programs).
- On OS/400 and Tandem NSK, this call is not supported.

```
MQBACK (Hconn, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_ENVIRONMENT_ERROR
(2012, X'7DC') Call not valid in environment.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.

MQRC_OUTCOME_MIXED
(2123, X'84B') Result of commit or back-out operation is mixed.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

See Chapter 5, “Return codes” on page 383 for more details.

Usage notes

1. This call is available only in those environments where there is no suitable unit-of-work manager; in these cases the queue manager itself coordinates the units of work. These can be:
   - A local unit of work, where the changes affect only MQ resources.
   - A global unit of work, where the changes can affect resources belonging to other resource managers, as well as affecting MQ resources.

   See “MQBEGIN – Begin unit of work” on page 244 for further details about local and global units of work.

   In environments where there is a suitable unit-of-work manager, the appropriate back-out call must be used instead of MQBACK, or the application terminated abnormally in order to back out the unit of work.
   - On MVS/ESA, this call is used only by batch programs (including IMS batch DL/I programs). It is not supported for CICS applications, which should use the EXEC CICS SYNCPOINT ROLLBACK command instead to cause changes to be backed out. IMS applications (other than batch DL/I programs) should use IMS calls, such as ROLB.
   - On OS/400 and Tandem NSK, this call is not supported.
2. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:
   - The values of the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD.
   - Whether the message is part of a unit of work.
   - For the MQPUT call: whether the message is persistent or nonpersistent.

   The queue manager keeps *three* sets of group and segment information, one set for each of the following:
   - The last successful MQPUT call (this can be part of a unit of work).
   - The last successful MQGET call that removed a message from the queue (this can be part of a unit of work).
   - The last successful MQGET call that browsed a message on the queue (this *cannot* be part of a unit of work).

If the application puts or gets the messages as part of a unit of work, and the application then decides to back out the unit of work, the group and segment information is restored to the value that it had previously:

- The information associated with the MQPUT call is restored to the value that it had prior to the first successful MQPUT call for that queue handle in the current unit of work.
- The information associated with the MQGET call is restored to the value that it had prior to the first successful MQGET call for that queue handle in the current unit of work.

Queues which were updated by the application after the unit of work had started, but outside the scope of the unit of work, do not have their group and segment information restored if the unit of work is backed out.

Restoring the group and segment information to its previous value when a unit of work is backed out allows the application to spread a large message group or large logical message consisting of many segments across several units of work, and to restart at the correct point in the message group or logical message if one of the units of work fails. Using several units of work may be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point in the event that a system failure occurs. For details of how to restart at the correct point after a system failure, see the MQPMO_LOGICAL_ORDER option described in "MQPMO – Put message options" on page 173, and the MQGMO_LOGICAL_ORDER option described in "MQGMO – Get-message options" on page 56.

The remaining usage notes apply only when the queue manager coordinates the units of work:

3. A unit of work has the same scope as a connection handle. This means that all MQ calls which affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See the *Hconn* parameter described in "MQCONN – Connect queue manager" on page 261 for information about the scope of connection handles.
4. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
5. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but which never issues a commit or backout call, will cause queues to fill up with messages that are not available to other applications.
6. On MVS/ESA, ending an application abnormally while there are uncommitted requests causes an implicit backout to occur.
7. On Tandem NSK, MQBACK always returns a *CompCode* of MQCC_FAILED and a *Reason* of MQRC_ENVIRONMENT_ERROR. Transactions are managed externally through TM/MP.

C language invocation

```
MQBACK (Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;     /* Connection handle */
MQLONG   CompCode;  /* Completion code */
MQLONG   Reason;    /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQBACK' USING HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON    PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2 and, Windows NT)

```
call MQBACK (Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn     fixed bin(31);  /* Connection handle */
dcl CompCode  fixed bin(31);  /* Completion code */
dcl Reason    fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
         CALL MQBACK,(HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT Hconn;
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL  MQBACK(HConn, CC, Reason);
```

MQBEGIN – Begin unit of work

The MQBEGIN call begins a unit of work that is coordinated by the queue manager, and that may involve external resource managers.

This call is supported in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

```
MQBEGIN (Hconn, BeginOptions, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

BeginOptions (MQBO) – input/output
Options that control the action of MQBEGIN.

See “MQBO – Begin options” on page 19 for details.

BeginOptions is a reserved parameter. Programs written in C or S/390 assembler can specify a null parameter address, instead of specifying the address of an MQBO structure.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_NO_EXTERNAL_PARTICIPANTS
(2121, X'849') No participating resource managers registered.
MQRC_PARTICIPANT_NOT_AVAILABLE
(2122, X'84A') Participating resource manager not available.

If *CompCode* is MQCC_FAILED:

MQRC_BO_ERROR
(2134, X'856') Begin-options structure not valid.

MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_ENVIRONMENT_ERROR
(2012, X'7DC') Call not valid in environment.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.
MQRC_UOW_IN_PROGRESS
(2128, X'850') Unit of work already started.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. The MQBEGIN call can be used to start a unit of work that is coordinated by the queue manager and that may involve changes to resources owned by other resource managers.

   The queue manager supports three types of unit-of-work:

   Queue-manager-coordinated local unit of work
   This is a unit of work in which the queue manager is the only participant, and so the queue manager acts as the unit-of-work coordinator.

   - To start this type of unit of work, the MQPMO_SYNCPOINT or MQGMO_SYNCPOINT option should be specified on the first MQPUT, MQPUT1, or MQGET call in the unit of work. It is not necessary for the application to issue the MQBEGIN call to start the unit of work. However, if it is used the unit of work is started, but the call completes with MQCC_WARNING and reason code MQRC_NO_EXTERNAL_PARTICIPANTS.
   - To commit or back out this type of unit of work, the MQCMIT and MQBACK calls must be used. If the application issues neither call, the unit of work is committed if the application issues the MQDISC call, but backed out if the application ends without issuing the MQDISC call.

   Queue-manager-coordinated global unit of work
   This is a unit of work in which the queue manager acts as the unit-of-work coordinator, both for MQ resources *and* for resources belonging to other resource managers. Those resource managers cooperate with the queue manager to ensure that all changes to resources in the unit of work are committed or backed out together.

   - To start this type of unit of work, the MQBEGIN call must be used.

- To commit or back out this type of unit of work, the MQCMIT and MQBACK calls must be used. If the application issues neither call, the unit of work is committed if the application issues the MQDISC call, but backed out if the application ends without issuing the MQDISC call.

Externally-coordinated global unit of work

This is a unit of work in which the queue manager is a participant, but the queue manager does not act as the unit-of-work coordinator. Instead, there is an external unit-of-work coordinator with whom the queue manager cooperates.

- To start this type of unit of work, the relevant call provided by the external unit-of-work coordinator must be used. If the MQBEGIN call is used to try to start the unit of work, the call fails with reason code MQRC_ENVIRONMENT_ERROR.
- To commit or back out this type of unit of work, the commit and back-out calls provided by the external unit-of-work coordinator must be used; the MQCMIT and MQBACK calls cannot be used.

2. An application can participate in only one unit of work at a time. The MQBEGIN call fails with reason code MQRC_UOW_IN_PROGRESS if there is already a unit of work in existence for the application, regardless of which type of unit of work it is.
3. The MQBEGIN call is not valid in an MQ client environment. An attempt to use the call fails with reason code MQRC_ENVIRONMENT_ERROR.
4. When the queue manager is acting as the unit-of-work coordinator for global units of work, the resource managers that can participate in the unit of work are defined in the queue manager's configuration file.

C language invocation

```
MQBEGIN (Hconn, &BeginOptions, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;         /* Connection handle */
MQBO     BeginOptions;  /* Options that control the action of MQBEGIN */
MQLONG   CompCode;      /* Completion code */
MQLONG   Reason;        /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQBEGIN' USING HCONN, BEGINOPTIONS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN         PIC S9(9) BINARY.
**   Options that control the action of MQBEGIN
 01  BEGINOPTIONS.
     COPY CMQBOV.
**   Completion code
 01  COMPCODE      PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON        PIC S9(9) BINARY.
```

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

```
call MQBEGIN (Hconn, BeginOptions, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn         fixed bin(31);  /* Connection handle */
dcl BeginOptions  like MQBO;      /* Options that control the action of
                                     MQBEGIN */
dcl CompCode      fixed bin(31);  /* Completion code */
dcl Reason        fixed bin(31);  /* Reason code qualifying CompCode */
```

MQCLOSE – Close object

The MQCLOSE call relinquishes access to an object, and is the inverse of the MQOPEN call.

```
MQCLOSE (Hconn, Hobj, Options, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

Hobj (MQHOBJ) – input/output
Object handle.

This handle represents the object that is being closed. The object can be of any type. 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 for the environment. This value is:

MQHO_UNUSABLE_HOBJ
Unusable object handle.

On MVS/ESA, *Hobj* is set to a value that is undefined.

Options (MQLONG) – input
Options that control the action of MQCLOSE.

The *Options* parameter controls how the object is closed. Only permanent dynamic queues can be closed in more than one way, being either retained or deleted; these are queues whose *DefinitionType* attribute has the value MQQDT_PERMANENT_DYNAMIC (see the *DefinitionType* attribute described in “Attributes for local queues and model queues” on page 348). The close options are summarized in Table 59 on page 250.

One (and only one) of the following must be specified:

MQCO_NONE
No optional close processing required.

This *must* be specified for:

- Objects other than queues
- Predefined queues

- Temporary dynamic queues (but only in those cases where *Hobj* is *not* the handle returned by the MQOPEN call that created the queue).
- Distribution lists

In all of the above cases, the object is retained and not deleted.

If this option is specified for a temporary dynamic queue:

- The queue is deleted, if it was created by the MQOPEN call that returned *Hobj*; any messages that are on the queue are purged.
- In all other cases the queue (and any messages on it) are retained.

If this option is specified for a permanent dynamic queue, the queue is retained and not deleted.

On MVS/ESA, if the queue is a dynamic queue that has been logically deleted (see Usage note 3 on page 252), and this is the last handle for it, the queue is physically deleted.

MQCO_DELETE
Delete the queue.

The queue is deleted if either of the following is true:

- It is a permanent dynamic queue, and there are no messages on the queue and no uncommitted get or put requests outstanding for the queue (either for the current task or any other task).
- It is the temporary dynamic queue that was created by the MQOPEN call that returned *Hobj*. In this case, all the messages on the queue are purged.

In all other cases the call fails with reason code MQRC_OPTION_NOT_VALID_FOR_TYPE, and the object is not deleted.

On MVS/ESA, if the queue is a dynamic queue that has been logically deleted (see Usage note 3 on page 252), and this is the last handle for it, the queue is physically deleted.

MQCO_DELETE_PURGE
Delete the queue, purging any messages on it.

The queue is deleted if either of the following is true:

- It is a permanent dynamic queue and there are no uncommitted get or put requests outstanding for the queue (either for the current task or any other task).
- It is the temporary dynamic queue that was created by the MQOPEN call that returned *Hobj*.

In all other cases the call fails with reason code MQRC_OPTION_NOT_VALID_FOR_TYPE, and the object is not deleted.

Table 59. Effect of MQCLOSE options on various types of object and queue. This table shows which close options are valid, and whether the object is retained or deleted.

Type of object or queue	MQCO_NONE	MQCO_DELETE	MQCO_DELETE_PURGE
Object other than a queue	retained	not valid	not valid
Predefined queue	retained	not valid	not valid
Permanent dynamic queue	retained	deleted if empty and no pending updates	messages deleted; queue deleted if no pending updates
Temporary dynamic queue (call issued by creator of queue)	deleted	deleted	deleted
Temporary dynamic queue (call not issued by creator of queue)	retained	not valid	not valid
Distribution list	retained	not valid	not valid

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.
MQRC_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.
MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_OPTION_NOT_VALID_FOR_TYPE
(2045, X'7FD') Option not valid for object type.
MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_Q_NOT_EMPTY
(2055, X'807') Queue contains one or more messages or uncommitted put or get requests.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SECURITY_ERROR
(2063, X'80F') Security error occurred.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

See Chapter 5, “Return codes” on page 383 for more details.

Usage notes

1. When an application issues the MQDISC call, or ends either normally or abnormally, any objects that were opened by the application and are still open are closed automatically with the MQCO_NONE option.
2. The following points apply only if the object being closed is a *queue*:
   - If operations on the queue were performed as part of a unit of work, the queue can be closed before or after the syncpoint occurs without affecting the outcome of the syncpoint.
   - On MVS/ESA, if there is an MQGET request with the MQGMO_SET_SIGNAL option outstanding against the queue handle being closed, the request is canceled (see the MQGMO_SET_SIGNAL option described in “MQGMO – Get-message options” on page 56). Signal

requests for the same queue but lodged against different handles (*Hobj*) are not affected (unless it is a dynamic queue that is being deleted, in which case they are also canceled).

- If the queue was opened with the MQOO_BROWSE option, the browse cursor is destroyed. If the queue is subsequently reopened with the MQOO_BROWSE option, a new browse cursor is created (see the MQOO_BROWSE option described in MQOPEN).
- If a message is currently locked for this handle at the time of the MQCLOSE call, the lock is released (see the MQGMO_LOCK option described in "MQGMO – Get-message options" on page 56).

3. The following points apply only if the object being closed is a *dynamic queue*:
   - For a dynamic queue, the options MQCO_DELETE or MQCO_DELETE_PURGE can be specified regardless of the options specified on the corresponding MQOPEN call.
   - When a temporary dynamic queue is closed using the *Hobj* handle returned by the MQOPEN call that created it, the queue is deleted (along with any messages that may still be on it) regardless of which of the valid options is specified in the *Options* parameter. This is true even if there are uncommitted MQGET, MQPUT, or MQPUT1 calls (issued using this or another handle) outstanding against the queue; any uncommitted updates that are lost *do not* cause the unit of work of which they are a part to fail.
   - When a dynamic queue is deleted, any MQGET requests with the MQGMO_WAIT option that are outstanding against the queue (using different *Hobj* handles) are canceled and reason code MQRC_Q_DELETED is returned. See the MQGMO_WAIT option described in "MQGMO – Get-message options" on page 56.
   - After a dynamic queue (either temporary or permanent) has been deleted, any call (other than MQCLOSE) that attempts to reference the queue using another previously acquired *Hobj* handle will fail with reason code MQRC_Q_DELETED.
   - On MVS/ESA, until the last such handle has been closed, the queue is logically deleted, but does still exist (for example, it can still be displayed), although no messages can be retrieved from it or put on it. During this time, any attempt to create a new queue (either dynamic or predefined) with the same name fails; in the case of a dynamic queue the MQOPEN call fails with the reason code MQRC_NAME_IN_USE. This is true for the application that caused the queue to become logically deleted, as well as for other applications.

     After the last *Hobj* handle referencing the queue has been closed, the queue is physically deleted, and a new queue with the same name can now be created. However, in the case of a temporary dynamic queue, if there are any corresponding unresolved units of work when the last *Hobj* handle referencing the queue has been closed, the queue is not physically deleted until the application terminates.

     As a result of a race condition, it is possible that a logically-deleted permanent dynamic queue does have uncommitted updates. In this case the queue can only be physically deleted after the corresponding units of work have been resolved (as well as all of the handles closed).

- When an MQCLOSE call is issued to delete a permanent dynamic queue, using an *Hobj* handle other than the one returned by the MQOPEN call that created the queue, a check is made that the user identifier which was used to validate the MQOPEN call (the alternate user identifier if MQOO_ALTERNATE_USER_AUTHORITY was specified) is authorized to delete the queue.

  No check is made when a temporary dynamic queue is deleted in this way, nor for a permanent dynamic queue if the handle specified is the one returned by the MQOPEN call that created the queue.

4. The following points apply only if the object being closed is a *distribution list*:

   - The only valid close option for a distribution list is MQCO_NONE; the call fails with reason code MQRC_OPTIONS_ERROR or MQRC_OPTION_NOT_VALID_FOR_TYPE if any other options are specified.
   - When a distribution list is closed, individual completion codes and reason codes are not returned for the queues in the list – only the *CompCode* and *Reason* parameters of the call are available for diagnostic purposes.

     If a failure occurs closing one of the queues, the queue manager continues processing and attempts to close the remaining queues in the distribution list. The *CompCode* and *Reason* parameters of the call are then set to return information describing the failure. Thus it is possible for the completion code to be MQCC_FAILED, even though most of the queues were closed successfully. The queue that encountered the error is not identified.

     If there is a failure on more than one queue, it is not defined which failure is reported in the *CompCode* and *Reason* parameters.

5. On OS/400, if the application was connected implicitly when the first MQOPEN call was issued, an implicit MQDISC occurs when the last MQCLOSE is issued.

C invocation

```
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 */
```

COBOL language invocation

```
     CALL 'MQCLOSE' USING HCONN, HOBJ, OPTIONS, COMPCODE,
                          REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Object handle
 01  HOBJ      PIC S9(9) BINARY.
**   Options that control the action of MQCLOSE
 01  OPTIONS   PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON    PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQCLOSE (Hconn, Hobj, Options, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn     fixed bin(31);  /* Connection handle */
dcl Hobj      fixed bin(31);  /* Object handle */
dcl Options   fixed bin(31);  /* Options that control the action of
                                 MQCLOSE */
dcl CompCode  fixed bin(31);  /* Completion code */
dcl Reason    fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
          CALL MQCLOSE,(HCONN,HOBJ,OPTIONS,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
HOBJ                DS      F           Object handle
OPTIONS             DS      F           Options that control the action
*                                       of MQCLOSE
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn ;
INT(32)   .EXT HObj;
INT(32)   Options;
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL MQCLOSE(HConn, HObj, Options, CC, Reason);
```

MQCMIT – Commit changes

The MQCMIT call indicates to the queue manager that the application has reached a syncpoint, and that all of the message gets and puts that have occurred since the last syncpoint are to be made permanent. Messages put as part of a unit of work are made available to other applications; messages retrieved as part of a unit of work are deleted.

- On MVS/ESA, the call is used only by batch programs (including IMS batch DL/I programs).
- On OS/400 and Tandem NSK, this call is not supported.

```
MQCMIT (Hconn, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_BACKED_OUT
(2003, X'7D3') Unit of work encountered fatal error or backed out.
MQRC_OUTCOME_PENDING
(2124, X'84C') Result of commit operation is pending.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.

MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_ENVIRONMENT_ERROR
(2012, X'7DC') Call not valid in environment.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_OUTCOME_MIXED
(2123, X'84B') Result of commit or back-out operation is mixed.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

See Chapter 5, “Return codes” on page 383 for more details.

Usage notes

1. This call is available only in those environments where there is no suitable unit-of-work manager; in these cases the queue manager itself coordinates the units of work. These can be:

   - A local unit of work, where the changes affect only MQ resources.
   - A global unit of work, where the changes can affect resources belonging to other resource managers, as well as affecting MQ resources.

   See “MQBEGIN – Begin unit of work” on page 244 for further details about local and global units of work.

   In environments where there *is* a suitable unit-of-work manager, the appropriate commit call must be used instead of MQCMIT. The environment may also support an implicit syncpoint caused by the application terminating normally.

   - On MVS/ESA, this call is used only by batch programs (including IMS batch DL/I programs). It is not supported for CICS applications, which should use the EXEC CICS SYNCPOINT command instead to cause a syncpoint, or end the transaction, and thus cause an implicit syncpoint. IMS applications (other than batch DL/I programs) should use IMS calls such as GU and CHKP.
   - On OS/400 and Tandem NSK, this call is not supported.

2. When an application puts or gets messages in groups or segments of logical messages, the queue manager retains information relating to the message group and logical message for the last successful MQPUT and MQGET calls. This information is associated with the queue handle, and includes such things as:

   - The values of the *GroupId*, *MsgSeqNumber*, *Offset*, and *MsgFlags* fields in MQMD.

- Whether the message is part of a unit of work.
- For the MQPUT call: whether the message is persistent or nonpersistent.

When a unit of work is committed, the queue manager retains the group and segment information, and the application can continue putting or getting messages in the current message group or logical message.

Retaining the group and segment information when a unit of work is committed allows the application to spread a large message group or large logical message consisting of many segments across several units of work. Using several units of work may be advantageous if the local queue manager has only limited queue storage. However, the application must maintain sufficient information to be able to restart putting or getting messages at the correct point in the event that a system failure occurs. For details of how to restart at the correct point after a system failure, see the MQPMO_LOGICAL_ORDER option described in "MQPMO – Put message options" on page 173, and the MQGMO_LOGICAL_ORDER option described in "MQGMO – Get-message options" on page 56.

The remaining usage notes apply only when the queue manager coordinates the units of work:

3. A unit of work has the same scope as a connection handle. This means that all MQ calls which affect a particular unit of work must be performed using the same connection handle. Calls issued using a different connection handle (for example, calls issued by another application) affect a different unit of work. See the *Hconn* parameter described in MQCONN for information about the scope of connection handles.
4. Only messages that were put or retrieved as part of the current unit of work are affected by this call.
5. If an application ends without issuing the MQCMIT or MQBACK call when there are uncommitted changes within a unit of work, the disposition of those changes depends on how the application ends:
   - If the application issues the MQDISC call before ending, that call causes the unit of work to be committed.

     Note: On MVS/ESA, the MQDISC call has this effect only for batch applications (including IMS and batch DL/1 applications). For CICS applications, the MQDISC call does not commit the unit of work.
   - If the application *does not* issue the MQDISC call but otherwise ends normally, the action taken depends on the environment:
     – On MVS/ESA, the unit of work is committed.
     – In all other environments, the unit of work is backed out.

     Because of the differences between environments, applications which are intended to be portable should always issue the MQCMIT or MQDISC call to commit the unit of work before ending, or the MQBACK call to back out the unit of work.
   - If the application ends abnormally, the unit of work is backed out; this has the same effect as the application issuing the MQBACK call.
6. A long-running application that issues MQGET, MQPUT, or MQPUT1 calls within a unit of work, but which never issues a commit or back-out call, will

cause queues to fill up with messages that are not available to other applications.

7. Note that in some environments, if the *Reason* parameter is MQRC_CONNECTION_BROKEN (with a *CompCode* of MQCC_FAILED), it is possible that the unit of work was successfully committed.

   This applies to MQ client applications running in the following environments: OpenVMS, OS/2, Tandem NSK, UNIX systems, and Windows NT.

8. On Tandem NSK, the MQCMIT call always returns a *CompCode* of MQCC_FAILED and a *Reason* of MQRC_ENVIRONMENT_ERROR. Transactions are managed externally through TM/MP.

C language invocation

```
MQCMIT (Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;     /* Connection handle */
MQLONG   CompCode;  /* Completion code */
MQLONG   Reason;    /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQCMIT' USING HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON    PIC S9(9) BINARY.
```

PL/I invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQCMIT (Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn     fixed bin(31);  /* Connection handle */
dcl CompCode  fixed bin(31);  /* Completion code */
dcl Reason    fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
         CALL MQCMIT,(HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT Hconn;
INT(32)   .EXT CC;
INT(32)   .EXT Reason;
```

MQCONN – Connect queue manager

The MQCONN call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent message queuing calls.

- On MVS/ESA, this call need not be issued by CICS applications. These applications are connected automatically to the queue manager to which the CICS system is connected. However, the MQCONN and MQDISC calls are still accepted from CICS applications.
- On OS/400, this call need not be issued. Applications are connected automatically to the queue manager when they issue the first MQOPEN call. However, the MQCONN and MQDISC calls are still accepted from OS/400 applications.

```
MQCONN (QMgrName, Hconn, CompCode, Reason)
```

Parameters

QMgrName (MQCHAR48) – input

Name of queue manager.

The name specified must be the name of a *connectable* queue manager. 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. If the name consists entirely of blanks, the name of the *default* queue manager is used.

The queue managers to which it is possible to connect are determined by the environment:

- On MVS/ESA:
  - For CICS, you can use only the queue manager to which the CICS system is connected.
  - For IMS, only queue managers which are listed in the subsystem definition table (CSQQDEFV), *and* listed in the SSM table in IMS, are connectable (see Usage note 6 on page 265).
  - For MVS batch and TSO, only queue managers that reside on the same system as the application are connectable (see Usage note 6 on page 265).
- On OS/400, only the default queue manager is connectable.

MQ client applications: For MQ client applications, a connection is attempted for each client-connection channel definition with the specified queue-manager name, until one is successful. The queue manager, however, must have the same name as the specified name. If an all-blank name is specified, each client-connection channel with an all-blank queue-manager name is tried until one is successful; in this case there is no check against the actual name of the queue manager.

MQ client applications are not supported in the following environments: MVS/ESA, OS/400, 16-bit Windows, 32-bit Windows. However, MVS/ESA

and OS/400 can act as MQ servers, to which MQ client applications can connect.

Queue-manager groups: If the specified name starts with an asterisk (*), the actual queue manager to which connection is made may have a name that is different from that specified by the application. The specified name (without the asterisk) defines a *group* of queue managers that are eligible for connection. The implementation selects one from the group by trying each one in turn (in no defined order) until one is found to which a connection can be made. If none of the queue managers in the group is available for connection, the call fails. Each queue manager is tried once only. If an asterisk alone is specified for the name, an implementation-defined default queue-manager group is used.

Queue-manager groups are supported only for applications running in a client environment; the call fails if a non-client application specifies a queue-manager name beginning with an asterisk. A group is defined by providing several client connection channel definitions with the same queue-manager name (the specified name without the asterisk), to communicate with each of the queue managers in the group. The default group is defined by providing one or more client connection channel definitions, each with a blank queue-manager name (specifying an all-blank name therefore has the same effect as specifying a single asterisk for the name for a client application).

After connecting to one queue manager of a group, an application can specify blanks in the usual way in the queue-manager name fields in the message and object descriptors to mean the name of the queue manager to which the application has actually connected (the *local queue manager*). If the application needs to know this name, the MQINQ call can be issued to inquire the *QMgrName* queue-manager attribute.

Prefixing an asterisk to the connection name in this way implies that the application is not sensitive to which queue manager in the group the application is connected. This will not be suitable for certain types of application, for example those which need to get messages from a particular queue at a particular queue manager; such applications should not prefix the name with an asterisk. Use of queue-manager groups *is* suitable for applications that put messages, and/or get messages from temporary dynamic queues which they have created.

Note that if an asterisk is specified, the maximum length of the remainder of the name is 47 characters.

The length of this parameter is given by MQ_Q_MGR_NAME_LENGTH. Queue-manager groups are not supported in the following environments: MVS/ESA, OS/400, 16-bit Windows, 32-bit Windows.

Hconn (MQHCONN) – 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 unit of processing that defines the scope of the handle terminates.

The scope of the handle is restricted to the smallest unit of parallel processing within the environment concerned; the handle is not valid outside the unit of parallel processing from which the MQCONN call was issued.

- On OpenVMS, the scope of the handle is the thread issuing the call.
- On DOS client, the scope of the handle is the system.
- On MVS/ESA, the scope of the handle is:
  - For CICS: the CICS task
  - For IMS: the Task Control Block, excluding any subtasks (usually this is the application program running in the dependent region)
  - For MVS batch and TSO: the Task Control Block, excluding any subtasks issuing the call. For IMS and MVS batch applications, the scope of the handle excludes any subtasks of the task.
- On OS/2, the scope of the handle is the thread issuing the call.
- On OS/400, the scope of the handle is the job issuing the call.
- On Tandem NSK, the scope of the handle is the thread issuing the call.
- On UNIX systems, the scope of the handle is the thread issuing the call.
- On Windows client and 16-bit Windows, the scope of the handle is the process issuing the call.
- On 32-bit Windows and Windows NT, the scope of the handle is the thread issuing the call.

On OS/400, and on MVS/ESA for CICS applications, the value returned is:

MQHC_DEF_HCONN
: Default connection handle.

CompCode (MQLONG) – output
: Completion code.

It is one of the following:

MQCC_OK
: Successful completion.

MQCC_WARNING
: Warning (partial completion).

MQCC_FAILED
: Call failed.

Reason (MQLONG) – output
: Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
: (0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_ALREADY_CONNECTED
: (2002, X'7D2') Application already connected.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_CONN_LOAD_ERROR
: (2129, X'851') Unable to load adapter connection module.

MQRC_ADAPTER_DEFS_ERROR
(2131, X'853') Adapter subsystem definition module not valid.
MQRC_ADAPTER_DEFS_LOAD_ERROR
(2132, X'854') Unable to load adapter subsystem definition module.
MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_ADAPTER_STORAGE_SHORTAGE
(2127, X'84F') Insufficient storage for adapter.
MQRC_ANOTHER_Q_MGR_CONNECTED
(2103, X'837') Another queue manager already connected.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CONN_ID_IN_USE
(2160, X'870') Connection identifier already in use.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_QUIESCING
(2202, X'89A') Connection quiescing.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_DUPLICATE_RECOV_COORD
(2163, X'873') Recovery coordinator already exists.
MQRC_MAX_CONNS_LIMIT_REACHED
(2025, X'7E9') Maximum number of connections reached.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SECURITY_ERROR
(2063, X'80F') Security error occurred.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. The queue manager to which connection is made using the MQCONN call is called the *local queue manager*.
2. Queues that belong to the local queue manager appear to the application as local queues. It is possible to put messages on and get messages from local queues.

   Queues belonging to remote queue managers appear as remote queues. It is possible to put messages on remote queues, but not possible to get messages from remote queues.
3. On MVS/ESA, this call must be issued by each batch or IMS application needing to use MQI calls.
4. After a failure of the queue manager, this call must be reissued. The application program can periodically reissue 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 the application is already connected, the same handle from the previous MQCONN call is returned, together with a warning completion code and reason code MQRC_ALREADY_CONNECTED.

   On MVS/ESA, this call need be reissued only by batch applications. IMS programs can keep reissuing the MQCONN call as many times as they want. However, this is not recommended for online message processing programs (MPPs).
5. Use the MQDISC call to disconnect from the queue manager.
6. On MVS/ESA, to define the available queue managers:
   - For batch applications, system programmers can use the CSQBDEF macro to create a module (CSQBDEFV) that defines the default queue-manager name.
   - For IMS applications, system programmers can use the CSQQDEFX macro to create a module (CSQQDEFV) that defines the names of the available queue managers and specifies the default queue manager.

   For more information on using these macros, see the *MQSeries for MVS/ESA System Management Guide*.
7. On MVS/ESA, a queue manager must be defined to the IMS control region and to each dependent region accessing that queue manager. To do this, you must create a subsystem member in the IMS.PROCLIB library and identify the subsystem member to the applicable IMS regions. If an application attempts to connect to a queue manager that is not defined in the subsystem member for its IMS region, the application abends.
8. On MVS/ESA, it is possible for batch, TSO, and IMS applications to connect to more than one queue manager concurrently.

C language invocation

```
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 */
```

COBOL language invocation

```
     CALL 'MQCONN' USING NAME, HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Name of queue manager
 01  NAME      PIC X(48).
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON    PIC S9(9) BINARY.
```

PL/I invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQCONN (Name, Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Name      char(48);       /* Name of queue manager */
dcl Hconn     fixed bin(31);  /* Connection handle */
dcl CompCode  fixed bin(31);  /* Completion code */
dcl Reason    fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
         CALL MQCONN,(NAME,HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
NAME                DS      CL48         Name of queue manager
HCONN               DS      F            Connection handle
COMPCODE            DS      F            Completion code
REASON              DS      F            Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
STRING   .EXT InQMgr[0:47];
INT(32)  .EXT HConn ;
INT(32)  .EXT CC;
INT(32)  .EXT Reason;

CALL  MQCONN(InQMgr, HConn, CC, Reason);
```

MQCONNX – Connect queue manager (extended)

The MQCONNX call connects an application program to a queue manager. It provides a queue manager connection handle, which is used by the application on subsequent MQ calls.

The MQCONNX call is similar to the MQCONN call, except that MQCONNX allows options to be specified to control the way that the call works.

This call is supported in the following environments: AIX, DOS client, HP-UX, OS/2, Sun Solaris, Windows client, Windows NT.

```
MQCONNX (QMgrName, ConnectOpts, Hconn, CompCode, Reason)
```

Parameters

QMgrName (MQCHAR48) – input
Name of queue manager.

See the *QMgrName* parameter described in "MQCONN – Connect queue manager" on page 261 for details.

ConnectOpts (MQCNO) – input/output
Options that control the action of MQCONNX.

See "MQCNO – Connect options" on page 35 for details.

Hconn (MQHCONN) – output
Connection handle.

See the *Hconn* parameter described in "MQCONN – Connect queue manager" on page 261 for details.

CompCode (MQLONG) – output
Completion code.

See the *CompCode* parameter described in "MQCONN – Connect queue manager" on page 261 for details.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

See the *Reason* parameter described in "MQCONN – Connect queue manager" on page 261 for details of possible reason codes. The following additional reason code can be returned by the MQCONNX call:

If *CompCode* is MQCC_FAILED:

MQRC_CNO_ERROR
(2139, X'85B') Connect-options structure not valid.

For more information on this reason code, see Chapter 5, "Return codes" on page 383.

C language invocation

```
MQCONNX (QMgrName, &ConnectOpts, &Hconn, &CompCode,
        &Reason);
```

Declare the parameters as follows:

```
MQCHAR48  QMgrName;     /* Name of queue manager */
MQCNO     ConnectOpts;  /* Options that control the action of MQCONNX */
MQHCONN   Hconn;        /* Connection handle */
MQLONG    CompCode;     /* Completion code */
MQLONG    Reason;       /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQCONNX' USING QMGRNAME, CONNECTOPTS, HCONN,
                          COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Name of queue manager
 01  QMGRNAME     PIC X(48).
**   Options that control the action of MQCONNX
 01  CONNECTOPTS.
     COPY CMQCNOV
**   Connection handle
 01  HCONN        PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE     PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON       PIC S9(9) BINARY.
```

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

```
call MQCONNX (QMgrName, ConnectOpts, Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl QMgrName     char(48);       /* Name of queue manager */
dcl ConnectOpts  like MQCNO;     /* Options that control the action of
                                    MQCONNX */
dcl Hconn        fixed bin(31);  /* Connection handle */
dcl CompCode     fixed bin(31);  /* Completion code */
dcl Reason       fixed bin(31);  /* Reason code qualifying CompCode */
```

MQDISC – Disconnect queue manager

The MQDISC call breaks the connection between the queue manager and the application program, and is the inverse of the MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, this call need not be issued. See “MQCONN – Connect queue manager” on page 261 for more information.

```
MQDISC (Hconn, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input/output
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, both the MQCONN and MQDISC calls can be omitted, and the following value used where *Hconn* would normally be specified:

MQHC_DEF_HCONN
Default connection handle.

On successful completion of the call, the queue manager sets *Hconn* to a value that is not a valid handle for the environment. This value is:

MQHC_UNUSABLE_HCONN
Unusable connection handle.

On MVS/ESA, *Hconn* is set to a value which is undefined.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output

Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_BACKED_OUT
(2003, X'7D3') Unit of work encountered fatal error or backed out.
MQRC_OUTCOME_PENDING
(2124, X'84C') Result of commit operation is pending.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_DISC_LOAD_ERROR
(2138, X'85A') Unable to load adapter disconnection module.
MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_OUTCOME_MIXED
(2123, X'84B') Result of commit or back-out operation is mixed.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. If an MQDISC call is issued when the application still has objects open, these objects are implicitly closed, with the close options set to MQCO_NONE.
2. On OS/400, and on MVS/ESA for CICS applications, this call need not be used; see the MQCONN call for more details.
3. On OpenVMS, OS/2, Tandem NSK, UNIX systems, and Windows NT, if a queue-manager-coordinated unit of work is in progress when this call is issued, an implicit syncpoint occurs; the unit of work is committed if possible. See MQBEGIN for more information about units of work coordinated by the queue manager.
4. On OS/2 and Windows NT, if an application terminates a thread without first issuing MQDISC, and a new thread is subsequently created (within the same process), and that thread issues message-queuing calls, the behavior of the queue manager is undefined.

C language invocation

```
MQDISC (&Hconn, &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;     /* Connection handle */
MQLONG   CompCode;  /* Completion code */
MQLONG   Reason;    /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQDISC' USING HCONN, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON    PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQDISC (Hconn, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn     fixed bin(31);  /* Connection handle */
dcl CompCode  fixed bin(31);  /* Completion code */
dcl Reason    fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
          CALL MQDISC,(HCONN,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn ;
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL MQDISC(HConn, CC, Reason);
```

MQGET – Get message

The MQGET call retrieves a message from a local queue that has been opened using the MQOPEN call.

```
MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer,
               DataLength, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

Hobj (MQHOBJ) – input
Object handle.

This handle represents the queue from which a message is to be retrieved. The value of *Hobj* was returned by a previous MQOPEN call. The queue must have been opened with one or more of the following options (see "MQOPEN – Open object" on page 297 for details):

MQOO_INPUT_SHARED
MQOO_INPUT_EXCLUSIVE
MQOO_INPUT_AS_Q_DEF
MQOO_BROWSE

MsgDesc (MQMD) – input/output
Message descriptor.

This structure describes the attributes of the message required, and the attributes of the message retrieved. See "MQMD – Message descriptor" on page 98 for details.

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 described in "MQGMO – Get-message options" on page 56).

If the application provides a version-1 MQMD, the message returned has an MQMDE prefixed to the application message data, but *only* if one or more of the fields in the MQMDE has a nondefault value. If all of the fields in the MQMDE have default values, the MQMDE is omitted. A format name of MQFMT_MD_EXTENSION in the *Format* field in MQMD indicates that an MQMDE is present.

GetMsgOpts (MQGMO) – input/output
Options that control the action of MQGET.

See “MQGMO – Get-message options” on page 56 for details.

BufferLength (MQLONG) – input
Length in bytes of the *Buffer* area.

Zero can be specified for messages that have no data, or if the message is to be removed from the queue and the data discarded (MQGMO_ACCEPT_TRUNCATED_MSG must be specified in this case).

Note: The length of the longest message that it is possible to read from the queue is given by the *MaxMsgLength* local queue attribute; see “Attributes for local queues and model queues” on page 348.

Buffer (MQBYTE×*BufferLength*) – 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*; this happens whether or not MQGMO_ACCEPT_TRUNCATED_MSG is specified on the *GetMsgOpts* parameter (see the *Options* field described in “MQGMO – Get-message options” on page 56 for more information).

The character set and encoding of the data in *Buffer* are given (respectively) by the *CodedCharSetId* and *Encoding* fields returned in the *MsgDesc* parameter. If these are different from the values required by the receiver, the receiver must convert the application message data to the character set and encoding required. The MQGMO_CONVERT option can be used with a user-written exit to perform the conversion of the message data (see “MQGMO – Get-message options” on page 56 for details of this option).

Note: All of the other parameters on the MQGET call are in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE, respectively).

If the call fails, the contents of the buffer may still have changed.

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 the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler can be null.

DataLength (MQLONG) – output
Length of the message.

This is the length in bytes of the application data *in the message*. If this is greater than *BufferLength*, only *BufferLength* bytes are returned in the *Buffer* parameter (that is, 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 described in "MQGMO – Get-message options" on page 56 for more information). This allows the application to determine the size of the buffer required to accommodate the message data, and then reissue the call with a buffer of the appropriate size.

However, if the MQGMO_CONVERT option is specified, and the converted message data is too long to fit in *Buffer*, the value returned for *DataLength* is:

- The length of the *unconverted* data, for queue-manager defined formats.

  In this case, if the nature of the data causes it to expand during conversion, the application must allocate a buffer somewhat bigger than the value returned by the queue manager for *DataLength*.
- The value returned by the data-conversion exit, for application-defined formats.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
: Successful completion.

MQCC_WARNING
: Warning (partial completion).

MQCC_FAILED
: Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

The reason codes listed below are the ones that the queue manager can return for the *Reason* parameter. If the application specifies the MQGMO_CONVERT option, and a user-written exit is invoked to convert some or all of the message data, it is the exit that decides what value is returned for the *Reason* parameter. As a result, values other than those documented below are possible.

If *CompCode* is MQCC_OK :

MQRC_NONE
: (0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_CONVERTED_MSG_TOO_BIG
: (2120, X'848') Converted message too big for application buffer.

MQRC_FORMAT_ERROR
: (2110, X'83E') Message format not valid.

MQRC_INCONSISTENT_CCSIDS
: (2243, X'8C3') Message segments have differing CCSIDs.

MQRC_INCONSISTENT_ENCODINGS
: (2244, X'8C4') Message segments have differing encodings.

MQRC_NO_MSG_LOCKED
: (2209, X'8A1') No message locked.

MQRC_NOT_CONVERTED
(2119, X'847') Application message data not converted.
MQRC_SIGNAL_REQUEST_ACCEPTED
(2070, X'816') No message returned (but signal request accepted).
MQRC_SOURCE_CCSID_ERROR
(2111, X'83F') Source coded character set identifier not valid.
MQRC_SOURCE_DECIMAL_ENC_ERROR
(2113, X'841') Packed-decimal encoding in message not recognized.
MQRC_SOURCE_FLOAT_ENC_ERROR
(2114, X'842') Floating-point encoding in message not recognized.
MQRC_SOURCE_INTEGER_ENC_ERROR
(2112, X'840') Source integer encoding not recognized.
MQRC_TARGET_CCSID_ERROR
(2115, X'843') Target coded character set identifier not valid.
MQRC_TARGET_DECIMAL_ENC_ERROR
(2117, X'845') Packed-decimal encoding specified by receiver not recognized.
MQRC_TARGET_FLOAT_ENC_ERROR
(2118, X'846') Floating-point encoding specified by receiver not recognized.
MQRC_TARGET_INTEGER_ENC_ERROR
(2116, X'844') Target integer encoding not recognized.
MQRC_TRUNCATED_MSG_ACCEPTED
(2079, X'81F') Truncated message returned (processing completed).
MQRC_TRUNCATED_MSG_FAILED
(2080, X'820') Truncated message returned (processing not completed).

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_CONV_LOAD_ERROR
(2133, X'855') Unable to load data conversion services modules.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_BACKED_OUT
(2003, X'7D3') Unit of work encountered fatal error or backed out.
MQRC_BUFFER_ERROR
(2004, X'7D4') Buffer parameter not valid.
MQRC_BUFFER_LENGTH_ERROR
(2005, X'7D5') Buffer length parameter not valid.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.

MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_QUIESCING
(2202, X'89A') Connection quiescing.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_DATA_LENGTH_ERROR
(2010, X'7DA') Data length parameter not valid.
MQRC_GET_INHIBITED
(2016, X'7E0') Gets inhibited for the queue.
MQRC_GMO_ERROR
(2186, X'88A') Get-message options structure not valid.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.
MQRC_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.
MQRC_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.
MQRC_INCONSISTENT_BROWSE
(2259, X'8D3') Inconsistent browse specification.
MQRC_INCONSISTENT_UOW
(2245, X'8C5') Inconsistent unit-of-work specification.
MQRC_INVALID_MSG_UNDER_CURSOR
(2246, X'8C6') Message under cursor not valid for retrieval.
MQRC_MATCH_OPTIONS_ERROR
(2247, X'8C7') Match options not valid.
MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.
| MQRC_MSG_SEQ_NUMBER_ERROR
| (2250, X'8CA') Message sequence number not valid.
MQRC_NO_MSG_AVAILABLE
(2033, X'7F1') No message available.
MQRC_NO_MSG_UNDER_CURSOR
(2034, X'7F2') Browse cursor not positioned on message.
MQRC_NOT_OPEN_FOR_BROWSE
(2036, X'7F4') Queue not open for browse.
MQRC_NOT_OPEN_FOR_INPUT
(2037, X'7F5') Queue not open for input.
MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.

MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SECOND_MARK_NOT_ALLOWED
(2062, X'80E') A message is already marked.
MQRC_SIGNAL_OUTSTANDING
(2069, X'815') Signal outstanding for this handle.
MQRC_SIGNAL1_ERROR
(2099, X'833') Signal field not valid.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_SYNCPOINT_LIMIT_REACHED
(2024, X'7E8') No more messages can be handled within current unit of work.
MQRC_SYNCPOINT_NOT_AVAILABLE
(2072, X'818') Syncpoint support not available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.
MQRC_UOW_NOT_AVAILABLE
(2255, X'8CF') Unit of work not available for the queue manager to use.
MQRC_WAIT_INTERVAL_ERROR
(2090, X'82A') Wait interval in MQGMO not valid.
MQRC_WRONG_GMO_VERSION
(2256, X'8D0') Wrong version of MQGMO supplied.
MQRC_WRONG_MD_VERSION
(2257, X'8D1') Wrong version of MQMD supplied.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. The message retrieved is normally deleted from the queue. This deletion can occur as part of the MQGET call itself, or as part of a syncpoint. Message deletion does not occur if an MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT option is specified on the *GetMsgOpts* parameter (see the *Options* field described in “MQGMO – Get-message options” on page 56).

   If the MQGMO_LOCK option is specified with one of the browse options, the browsed message is locked so that it is visible only to this handle.

   If the MQGMO_UNLOCK option is specified, a previously-locked message is unlocked. No message is retrieved in this case, and the *MsgDesc*, *BufferLength*, *Buffer* and *DataLength* parameters are not checked or altered.

2. If an application puts a sequence of messages on the same queue, the order of those messages is preserved provided that all of the following are true:

   - The messages all have the same priority.

- All of the MQPUT calls are made using the same object handle *Hobj*.

  In some environments, message sequence is also preserved when different object handles are used, provided the calls are made from the same application. The meaning of "same application" is determined by the environment:

  – On OpenVMS, the application is the thread.
  – On DOS client, the application is the system.
  – On MVS/ESA, the application is:
    - For CICS, the CICS task
    - For IMS, the Task Control Block
    - For MVS batch, the Task Control Block
  – On OS/2, the application is the thread.
  – On OS/400, the application is the job.
  – On Tandem NSK, the application is the thread.
  – On UNIX systems, the application is the thread.
  – On Windows client and 16-bit Windows, the application is the process.
  – On Windows NT and 32-bit Windows, the application is the thread.

- All of the MQPUT calls are within the same unit of work, or none of them is within a unit of work.
- The queue is local to the queue manager at which the MQPUT calls were made (but see note 2b).

If these conditions are satisfied, the messages will be presented to the receiving application in the order in which they were sent, provided that:

- The receiver does not deliberately change the order of retrieval, for example by specifying a particular *MsgId* or *CorrelId*.
- Only one receiver is getting messages from the queue.

  If there are two or more applications getting messages from the queue, they must agree with the sender the mechanism to be used to identify messages that belong to a sequence. For example, the sender could set all of the *CorrelId* fields in the messages in a sequence to a value that was unique to that sequence of messages.

Notes:

a. When messages are put onto a particular queue within a single unit of work, messages from other applications may be interspersed with the sequence of messages on the queue.

b. For remote queues, the order of the messages is preserved if the configuration is such that there is only one path from the sender's queue manager to the destination queue manager. If there is a possibility that some messages in the sequence may go on a different path (for example, because of reconfiguration, traffic balancing, or path selection based on message size), the order of the messages at the destination cannot be guaranteed. Messages destined for remote queues can also become out of sequence if one or more of them is put to a dead-letter queue (for example, because the destination queue is temporarily full).

If the required conditions are not met, applications can include their own sequencing information within the application message data, or establish a conversation scheme in which each message is acknowledged, and the acknowledgement received by the sender, before the next message is put.

3. Applications should test for the feedback code MQFB_QUIT in the *Feedback* field of the *MsgDesc* parameter. If this value is found, the application should end. See the *Feedback* field described in "MQMD – Message descriptor" on page 98 for more information.
4. If the queue identified by *Hobj* was opened with the MQOO_SAVE_ALL_CONTEXT option, and the completion code from the MQGET call is MQCC_OK or MQCC_WARNING, the context associated with the queue handle *Hobj* is set to the context of the message that has been retrieved (unless the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT option is set, in which case it is marked as not available). This context can be used on a subsequent MQPUT or MQPUT1 call (for example, when a message is forwarded to another queue). For more information on message context, see the *MQSeries Application Programming Guide*.
5. If the MQGMO_CONVERT option is included in the *GetMsgOpts* parameter, the application message data is converted to the representation requested by the receiving application, before the data is placed in the *Buffer* parameter.

   The *Format* field in the control information in the message identifies the structure of the application data, and the *CodedCharSetId* and *Encoding* fields in the control information in the message specify its character-set identifier and encoding. The application issuing the MQGET call specifies in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter the character-set identifier and encoding to which the application message data should be converted. If the *CodedCharSetId* and *Encoding* values in the control information in the message are identical to those in the *MsgDesc* parameter, no conversion is necessary.

   When conversion of the message data is necessary, the conversion is performed either by the queue manager itself or by a user-written exit, depending on the value of the *Format* field in the control information in the message:

   - The format names listed below are formats that are converted automatically by the queue manager; these are called "built-in" formats:

     MQFMT_ADMIN
     MQFMT_COMMAND_1
     MQFMT_COMMAND_2
     MQFMT_DEAD_LETTER_HEADER
     MQFMT_DIST_HEADER
     MQFMT_EVENT
     MQFMT_IMS
     MQFMT_IMS_VAR_STRING
     MQFMT_MD_EXTENSION
     MQFMT_PCF
     MQFMT_REF_MSG_HEADER
     MQFMT_STRING
     MQFMT_TRIGGER
     MQFMT_XMIT_Q_HEADER

   - The format name MQFMT_NONE is a special value that indicates that the nature of the data in the message is undefined. As a consequence, the queue manager does not attempt conversion when the message is retrieved from the queue.

Note: If MQGMO_CONVERT is specified on the MQGET call for a message that has a format name of MQFMT_NONE, and the character set or encoding of the message differs from that specified in the *MsgDesc* parameter, the message is still returned in the *Buffer* parameter (assuming no other errors), but the call completes with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

MQFMT_NONE can be used either when the nature of the message data means that it does not require conversion, or when the sending and receiving applications have agreed between themselves the form in which the message data should be sent.

- All other format names cause the message to be passed to a user-written exit for conversion. The exit has the same name as the format, apart from environment-specific additions. User-specified format names should not begin with the letters "MQ", as such names may conflict with queue-manager-defined format names supported in the future.

  See Appendix D, "Data-conversion" on page 495 for details of the data-conversion exit.

On return from MQGET, the following reason code indicates that the message was converted successfully:

MQRC_NONE

The following reason code indicates that the message *may* have been converted successfully; the application should check the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to find out:

MQRC_TRUNCATED_MSG_ACCEPTED

All other reason codes indicate that the message was not converted.

Note: The interpretation of the reason code described above will be true for conversions performed by user-written exits *only* if the exit conforms to the processing guidelines described in Appendix D, "Data-conversion" on page 495.

6. For the built-in formats listed above, the queue manager may perform *default conversion* of character strings in the message when the MQGMO_CONVERT option is specified. Default conversion allows the queue manager to use an installation-specified default character set that approximates the actual character set, when converting string data. As a result, the MQGET call can succeed with completion code MQCC_OK, instead of completing with MQCC_WARNING and reason code MQRC_SOURCE_CCSID_ERROR or MQRC_TARGET_CCSID_ERROR.

   Note: The result of using an approximate character set to convert string data is that some characters may be converted incorrectly. This can be avoided by using in the string only characters which are common to both the actual character set and the default character set.

   Default conversion applies both to the application message data and to character fields in the MQMD and MQMDE structures:

   - Default conversion of the application message data occurs only when *all* of the following are true:
     - The application specifies MQGMO_CONVERT.

  - The message contains data that must be converted either from or to a character set which is not supported.
  - Default conversion was enabled when the queue manager was installed or restarted.

- Default conversion of the character fields in the MQMD and MQMDE structures occurs as necessary, provided that default conversion is enabled for the queue manager. The conversion is performed even if the MQGMO_CONVERT option is not specified by the application on the MQGET call.

7. On Tandem NSK, the following restrictions apply:

   - The message retrieved by the MQGET call is deleted from the queue unless the MQGMO_BROWSE_FIRST option or the MQGMO_BROWSE_NEXT option is specified.
   - If MQGET is issued outside a Tandem TMF transaction *without* the MQGMO_NO_SYNCPOINT option, the reason code MQRC_UNIT_OF_WORK_NOT_STARTED is returned.
   - If the MQGMO_CONVERT option is specified for an MQGET call, and the message that is retrieved is not in one of the built-in formats (MQFMT_*), the message is passed to the data conversion exit function MQDATACONVEXIT() for conversion. A single data conversion exit is provided by the product, because the Tandem NSK operating system does not support dynamic linking. The format name of the unconverted message, from the MQMD of the message, is passed to MQDATACONVEXIT() in the *MsgDesc* parameter.

C language invocation

```
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 */
```

COBOL language invocation

```
     CALL 'MQGET' USING HCONN, HOBJ, MSGDESC, GETMSGOPTS,
                        BUFFERLENGTH, BUFFER, DATALENGTH, COMPCODE,
                        REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN          PIC S9(9) BINARY.
**   Object handle
 01  HOBJ           PIC S9(9) BINARY.
**   Message descriptor
 01  MSGDESC.
     COPY CMQMDV.
**   Options that control the action of MQGET
 01  GETMSGOPTS.
     COPY CMQGMOV.
**   Length in bytes of the Buffer area
 01  BUFFERLENGTH  PIC S9(9) BINARY.
**   Area to contain the message data
 01  BUFFER        PIC X(n).
**   Length of the message
 01  DATALENGTH    PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE      PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON        PIC S9(9) BINARY.
```

PL/I invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength, Buffer,
            DataLength, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn         fixed bin(31);  /* Connection handle */
dcl Hobj          fixed bin(31);  /* Object handle */
dcl MsgDesc       like MQMD;      /* Message descriptor */
dcl GetMsgOpts    like MQGMO;     /* Options that control the action of
                                     MQGET */
dcl BufferLength  fixed bin(31);  /* Length in bytes of the Buffer
                                     area */
dcl Buffer        char(n);        /* Area to contain the message data */
dcl DataLength    fixed bin(31);  /* Length of the message */
dcl CompCode      fixed bin(31);  /* Completion code */
dcl Reason        fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
      CALL MQGET,(HCONN,HOBJ,MSGDESC,GETMSGOPTS,BUFFERLENGTH,BUFFER,    X
            DATALENGTH,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
HOBJ                DS      F           Object handle
MSGDESC             CMQMDA              Message descriptor
GETMSGOPTS          CMQGMOA             Options that control the action
*                                       of MQGET
BUFFERLENGTH        DS      F           Length in bytes of the Buffer
*                                       area
BUFFER              DS      CL(n)       Area to contain the message data
DATALENGTH          DS      F           Length of the message
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT  Hconn;
INT(32)   .EXT  Hobj;
STRUCT    .EXT  MsgDesc(MQMD^Def);
STRUCT    .EXT  GetMsgOpt(MQGMO^Def);
INT(32)   .EXT  BufferLen;
INT(32)   .EXT  Buffer[0:BUFFER^LEN];
INT(32)   .EXT  CC;
INT(32)   .EXT  Reason;

CALL MQGET(HConn, HObj, MsgDesc, GetMsgOpt, BufferLen, Buffer,
                            DataLen, CC, Reason);
```

MQINQ – Inquire about object attributes

The MQINQ call returns an array of integers and a set of character strings containing the attributes of an object. The following types of object are valid:

- Queue
- Namelist (MVS/ESA only)
- Process definition
- Queue manager

```
MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
       CharAttrLength, CharAttrs, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call. call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

Hobj (MQHOBJ) – input
Object handle.

This handle represents the object (of any type) whose attributes are required. The handle must have been returned by a previous MQOPEN call that specified 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 number allowed is 256.

Selectors (MQLONG×*SelectorCount*) – 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, otherwise the call fails with completion code MQCC_FAILED and reason code MQRC_SELECTOR_ERROR.

In the special case of queues:

- If the selector is not valid for queues of *any* type, the call fails with completion code MQCC_FAILED and reason code MQRC_SELECTOR_ERROR.

- If the selector is applicable *only* to queues of type or types other than that of the object, the call succeeds with completion code MQCC_WARNING and reason code MQRC_SELECTOR_NOT_FOR_TYPE.

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 occur in *Selectors*. 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.

Notes:

1. The integer and character attribute selectors are allocated within two different ranges; the MQIA_★ selectors reside within the range MQIA_FIRST through MQIA_LAST, and the MQCA_★ selectors within the range MQCA_FIRST through MQCA_LAST.

   For each range, the constants MQIA_LAST_USED and MQCA_LAST_USED define the highest value that the queue manager will accept.

2. If all of the MQIA_★ selectors occur first, the same element numbers can be used to address corresponding elements in the *Selectors* and *IntAttrs* arrays.

3. If the *SelectorCount* parameter is zero, *Selectors* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler may be null.

For the MQCA_★ selectors in the following descriptions, the constant that defines the length in bytes of the resulting string in *CharAttrs* is given in parentheses.

Selectors for queue manager

MQCA_CHANNEL_AUTO_DEF_EXIT
: Automatic channel definition exit name (MQ_EXIT_NAME_LENGTH).

MQCA_COMMAND_INPUT_Q_NAME
: System command input queue name (MQ_Q_NAME_LENGTH).

MQCA_DEAD_LETTER_Q_NAME
: Name of dead-letter queue (MQ_Q_NAME_LENGTH).

MQCA_DEF_XMIT_Q_NAME
: Default transmission queue name (MQ_Q_NAME_LENGTH).

MQCA_Q_MGR_DESC
: Queue manager description (MQ_Q_MGR_DESC_LENGTH).

MQCA_Q_MGR_NAME
: Name of local queue manager (MQ_Q_MGR_NAME_LENGTH).

MQIA_AUTHORITY_EVENT
: Control attribute for authority events.

MQIA_CHANNEL_AUTO_DEF
: Control attribute for automatic channel definition.

MQIA_CHANNEL_AUTO_DEF_EVENT
: Control attribute for automatic channel definition events.

MQIA_CODED_CHAR_SET_ID
: Coded character set identifier.

MQIA_COMMAND_LEVEL
: Command level supported by queue manager.

MQIA_DIST_LISTS
: Distribution list support.

MQIA_INHIBIT_EVENT
: Control attribute for inhibit events.

MQIA_LOCAL_EVENT
: Control attribute for local events.

MQIA_MAX_HANDLES
: Maximum number of handles.

MQIA_MAX_MSG_LENGTH
: Maximum message length.

MQIA_MAX_PRIORITY
: Maximum priority.

MQIA_MAX_UNCOMMITTED_MSGS
: Maximum number of uncommitted messages within a unit of work.

MQIA_PERFORMANCE_EVENT
: Control attribute for performance events.

MQIA_PLATFORM
: Platform on which the queue manager resides.

MQIA_REMOTE_EVENT
: Control attribute for remote events.

MQIA_START_STOP_EVENT
: Control attribute for start stop events.

MQIA_SYNCPOINT
: Syncpoint availability.

MQIA_TRIGGER_INTERVAL
: Trigger interval.

On MVS/ESA, the following selectors are not supported:

MQCA_CHANNEL_AUTO_DEF_EXIT
MQIA_AUTHORITY_EVENT
MQIA_CHANNEL_AUTO_DEF
MQIA_CHANNEL_AUTO_DEF_EVENT
MQIA_DIST_LISTS
MQIA_INHIBIT_EVENT
MQIA_LOCAL_EVENT
MQIA_MAX_UNCOMMITTED_MSGS
MQIA_PERFORMANCE_EVENT
MQIA_REMOTE_EVENT
MQIA_START_STOP_EVENT

The selectors listed below are supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQCA_CHANNEL_AUTO_DEF_EXIT
MQIA_CHANNEL_AUTO_DEF
MQIA_CHANNEL_AUTO_DEF_EVENT
MQIA_DIST_LISTS

Selectors for namelists (MVS/ESA only)

MQCA_NAMELIST_DESC
: Namelist description (MQ_NAMELIST_DESC_LENGTH).

MQCA_NAMELIST_NAME
: Name of namelist object (MQ_NAMELIST_NAME_LENGTH).

MQCA_NAMES
: Name of each queue in the list (MQ_Q_NAME_LENGTH × Number of queue names in the list).

MQIA_NAME_COUNT
: Number of queue names in the list.

Selectors for all types of queue

MQCA_Q_DESC
: Queue description (MQ_Q_DESC_LENGTH).

MQCA_Q_NAME
: Queue name (MQ_Q_NAME_LENGTH).

MQIA_DEF_PERSISTENCE
: Default message persistence.

MQIA_DEF_PRIORITY
: Default message priority.

MQIA_INHIBIT_PUT
: Whether put operations are allowed.

MQIA_Q_TYPE
: Queue type.

Selectors for local queues

MQCA_BACKOUT_REQ_Q_NAME
: Excessive backout requeue name (MQ_Q_NAME_LENGTH).

MQCA_CREATION_DATE
: Queue creation date (MQ_CREATION_DATE_LENGTH).

MQCA_CREATION_TIME
: Queue creation time (MQ_CREATION_TIME_LENGTH).

MQCA_INITIATION_Q_NAME
: Initiation queue name (MQ_Q_NAME_LENGTH).

MQCA_PROCESS_NAME
: Name of process definition (MQ_PROCESS_NAME_LENGTH).

MQCA_STORAGE_CLASS
: Name of storage class (MQ_STORAGE_CLASS_LENGTH).

MQCA_TRIGGER_DATA
: Trigger data (MQ_TRIGGER_DATA_LENGTH).

MQIA_BACKOUT_THRESHOLD
: Backout threshold.

MQIA_CURRENT_Q_DEPTH
: Number of messages on queue.

MQIA_DEF_INPUT_OPEN_OPTION
: Default open-for-input option.

MQIA_DEFINITION_TYPE
: Queue definition type.

MQIA_DIST_LISTS
: Distribution list support.

MQIA_HARDEN_GET_BACKOUT
: Whether to harden backout count.

MQIA_INDEX_TYPE
: Type of index maintained for queue.

MQIA_INHIBIT_GET
: Whether get operations are allowed.

MQIA_MAX_MSG_LENGTH
: Maximum message length.

MQIA_MAX_Q_DEPTH
: Maximum number of messages allowed on queue.

MQIA_MSG_DELIVERY_SEQUENCE
: Whether message priority is relevant.

MQIA_OPEN_INPUT_COUNT
: Number of MQOPEN calls that have the queue open for input.

MQIA_OPEN_OUTPUT_COUNT
: Number of MQOPEN calls that have the queue open for output.

MQIA_Q_DEPTH_HIGH_EVENT
: Control attribute for queue depth high events.

MQIA_Q_DEPTH_HIGH_LIMIT
: High limit for queue depth.

MQIA_Q_DEPTH_LOW_EVENT
: Control attribute for queue depth low events.

MQIA_Q_DEPTH_LOW_LIMIT
: Low limit for queue depth.

MQIA_Q_DEPTH_MAX_EVENT
: Control attribute for queue depth max events.

MQIA_Q_SERVICE_INTERVAL
: Limit for queue service interval.

MQIA_Q_SERVICE_INTERVAL_EVENT
: Control attribute for queue service interval events.

MQIA_RETENTION_INTERVAL
: Queue retention interval.

MQIA_SCOPE
: Queue definition scope.

MQIA_SHAREABILITY
: Whether queue can be shared.

MQIA_TRIGGER_CONTROL
: Trigger control.

MQIA_TRIGGER_DEPTH
: Trigger depth.

MQIA_TRIGGER_MSG_PRIORITY
: Threshold message priority for triggers.

MQIA_TRIGGER_TYPE
: Trigger type.

MQIA_USAGE
: Usage.

On MVS/ESA, the following selectors are not supported:

MQIA_DIST_LISTS
MQIA_Q_DEPTH_HIGH_EVENT
MQIA_Q_DEPTH_HIGH_LIMIT
MQIA_Q_DEPTH_LOW_EVENT
MQIA_Q_DEPTH_LOW_LIMIT
MQIA_Q_DEPTH_MAX_EVENT
MQIA_Q_SERVICE_INTERVAL
MQIA_Q_SERVICE_INTERVAL_EVENT

MQIA_SCOPE

The following selectors are supported only on MVS/ESA:

MQCA_STORAGE_CLASS
MQIA_INDEX_TYPE

The selector listed below is supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQIA_DIST_LISTS

Selectors for local definitions of remote queues

MQCA_REMOTE_Q_MGR_NAME
: Name of remote queue manager (MQ_Q_MGR_NAME_LENGTH).

MQCA_REMOTE_Q_NAME
: Name of remote queue as known on remote queue manager (MQ_Q_NAME_LENGTH).

MQCA_XMIT_Q_NAME
: Transmission queue name (MQ_Q_NAME_LENGTH).

MQIA_SCOPE
: Queue definition scope.

On MVS/ESA, the following selector is not supported:

MQIA_SCOPE

Selectors for alias queues

MQCA_BASE_Q_NAME
: Name of queue that alias resolves to (MQ_Q_NAME_LENGTH).

MQIA_INHIBIT_GET
: Whether get operations are allowed.

MQIA_SCOPE
: Queue definition scope.

On MVS/ESA, the following selector is not supported:

MQIA_SCOPE

Selectors for process definitions

MQCA_APPL_ID
: Application identifier (MQ_PROCESS_APPL_ID_LENGTH).

MQCA_ENV_DATA
: Environment data (MQ_PROCESS_ENV_DATA_LENGTH).

MQCA_PROCESS_DESC
: Description of process definition (MQ_PROCESS_DESC_LENGTH).

MQCA_PROCESS_NAME
: Name of process definition (MQ_PROCESS_NAME_LENGTH).

MQCA_USER_DATA
: User data (MQ_PROCESS_USER_DATA_LENGTH).

MQIA_APPL_TYPE
: Application type.

IntAttrCount (MQLONG) – input
Count of integer attributes.

This is the number of elements in the *IntAttrs* array. Zero is a valid value.

If this is at least the number of MQIA_★ selectors in the *Selectors* parameter, all integer attributes requested are returned.

IntAttrs (MQLONG×*IntAttrCount*) – output
Array of integer attributes.

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 referred to; in this case, the parameter address passed by programs written in C or System/390 assembler may be null.

CharAttrLength (MQLONG) – input
Length of character attributes buffer.

This is the length in bytes of the *CharAttrs* parameter.

This must be at least the sum of the lengths of the requested character attributes (see *Selectors*). Zero is a valid value.

CharAttrs (MQCHAR×*CharAttrLength*) – 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 that needed to contain all of the requested character attributes (including padding), the bytes beyond the last attribute value returned are 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 *CharAttrs*.

If the *CharAttrLength* or *SelectorCount* parameter is zero, *CharAttrs* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler may be null.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_CHAR_ATTRS_TOO_SHORT
(2008, X'7D8') Not enough space allowed for character attributes.
MQRC_INT_ATTR_COUNT_TOO_SMALL
(2022, X'7E6') Not enough space allowed for integer attributes.
MQRC_SELECTOR_NOT_FOR_TYPE
(2068, X'814') Selector not applicable to queue type.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CHAR_ATTR_LENGTH_ERROR
(2006, X'7D6') Length of character attributes not valid.
MQRC_CHAR_ATTRS_ERROR
(2007, X'7D7') Character attributes string not valid.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.
MQRC_INT_ATTR_COUNT_ERROR
(2021, X'7E5') Count of integer attributes not valid.
MQRC_INT_ATTRS_ARRAY_ERROR
(2023, X'7E7') Integer attributes array not valid.

MQRC_NOT_OPEN_FOR_INQUIRE
(2038, X'7F6') Queue not open for inquire.
MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SELECTOR_COUNT_ERROR
(2065, X'811') Count of selectors not valid.
MQRC_SELECTOR_ERROR
(2067, X'813') Attribute selector not valid.
MQRC_SELECTOR_LIMIT_EXCEEDED
(2066, X'812') Count of selectors too big.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

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. When you open a model queue, even for inquiring about its attributes, a dynamic queue is created. The attributes of the dynamic queue (except for *CreationDate*, *CreationTime*, and *DefinitionType*) are the same as those of the model queue at the time the dynamic queue is created. If you subsequently use the MQINQ call with the same object handle, the queue manager returns the attributes of the dynamic queue, not those of the model queue.
3. The attributes returned by the MQINQ call directed at an alias queue are those of the alias queue, not those of the base queue to which the alias resolves.
4. If a number of attributes are to be inquired, and subsequently some of them are to be set using the MQSET call, it may be convenient to position at the beginning of the selector arrays the attributes that are to be set, so that the same arrays (with reduced counts) can be used for MQSET.

5. If more than one of the warning situations arise (see the *CompCode* parameter), the reason code returned is the *first* one in the following list that applies:
   a. MQRC_SELECTOR_NOT_FOR_TYPE
   b. MQRC_INT_ATTR_COUNT_TOO_SMALL
   c. MQRC_CHAR_ATTRS_TOO_SHORT
6. For more information about object attributes, see Chapter 4, “Attributes of MQSeries objects.”

C language invocation

```
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 */
```

COBOL language invocation

```
     CALL 'MQINQ' USING HCONN, HOBJ, SELECTORCOUNT,
                        SELECTORS-TABLE, INTATTRCOUNT, INTATTRS-TABLE,
                        CHARATTRLENGTH, CHARATTRS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN            PIC S9(9) BINARY.
**   Object handle
 01  HOBJ             PIC S9(9) BINARY.
**   Count of selectors
 01  SELECTORCOUNT    PIC S9(9) BINARY.
**   Array of attribute selectors
 01  SELECTORS-TABLE.
  02  SELECTORS        PIC S9(9) BINARY OCCURS n TIMES.
**   Count of integer attributes
 01  INTATTRCOUNT     PIC S9(9) BINARY.
**   Array of integer attributes
 01  INTATTRS-TABLE.
  02  INTATTRS         PIC S9(9) BINARY OCCURS n TIMES.
**   Length of character attributes buffer
 01  CHARATTRLENGTH   PIC S9(9) BINARY.
**   Character attributes
 01  CHARATTRS        PIC X(n).
**   Completion code
 01  COMPCODE         PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON           PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount,
            IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn           fixed bin(31);  /* Connection handle */
dcl Hobj            fixed bin(31);  /* Object handle */
dcl SelectorCount   fixed bin(31);  /* Count of selectors */
dcl Selectors(n)    fixed bin(31);  /* Array of attribute selectors */
dcl IntAttrCount    fixed bin(31);  /* Count of integer attributes */
dcl IntAttrs(n)     fixed bin(31);  /* Array of integer attributes */
dcl CharAttrLength  fixed bin(31);  /* Length of character attributes
                                       buffer */
dcl CharAttrs       char(n);        /* Character attributes */
dcl CompCode        fixed bin(31);  /* Completion code */
dcl Reason          fixed bin(31);  /* Reason code qualifying
                                       CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
      CALL MQINQ,(HCONN,HOBJ,SELECTORCOUNT,SELECTORS,INTATTRCOUNT,      X
            INTATTRS,CHARATTRLENGTH,CHARATTRS,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
HOBJ                DS      F           Object handle
SELECTORCOUNT       DS      F           Count of selectors
SELECTORS           DS      (n)F        Array of attribute selectors
INTATTRCOUNT        DS      F           Count of integer attributes
INTATTRS            DS      (n)F        Array of integer attributes
CHARATTRLENGTH      DS      F           Length of character attributes
*                                       buffer
CHARATTRS           DS      CL(n)       Character attributes
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn ;
INT(32)   .EXT HObj ;
INT(32)      SelectorCount;
INT(32)   .EXT Selectors[0:NUM^SELECTORS];
INT(32)      IntAttrCount;
INT(32)   .EXT IntAttrs[0:NUM^INT^ATTR];
INT(32)      CharAttrLength;
STRING    .EXT CharAttrs[0:LEN^CHAR^ATTR];
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

PROC MQINQ(HConn, HObj, SelectorCount, Selectors, IntAttrCount,
IntAttrs, CharAttrLength, CharAttrs, CC, Reason)
```

MQOPEN – Open object

The MQOPEN call establishes access to an object. The following types of object are valid:

- Queue (including distribution lists)
- Namelist (MVS/ESA only)
- Process definition (not 16-bit Windows, 32-bit Windows)
- Queue manager

```
MQOPEN (Hconn, ObjDesc, Options, Hobj, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

ObjDesc (MQOD) – input/output
Object descriptor.

This is a structure that identifies the object to be opened; see "MQOD – Object descriptor" on page 160 for details.

If the *ObjectName* field in the *ObjDesc* parameter is the name of a model queue, a dynamic local queue is created with the attributes of the model queue; this happens irrespective of the open options specified by the *Options* parameter. Subsequent operations using the *Hobj* returned by the MQOPEN call are performed on the new dynamic queue, and not on the model queue. This is true even for the MQINQ and MQSET calls. The name of the model queue in the *ObjDesc* parameter is replaced with the name of the dynamic queue created. The type of the dynamic queue is determined by the value of the *DefinitionType* attribute of the model queue (see "Attributes for local queues and model queues" on page 348). For information about the close options applicable to dynamic queues, see the description of the MQCLOSE call.

Options (MQLONG) – input
Options that control the action of MQOPEN.

At least one of the following options must be specified:

MQOO_BROWSE
MQOO_INPUT_★ (only one of these)
MQOO_INQUIRE
MQOO_OUTPUT
MQOO_SET

See below for details of these options; other options can be specified as required. If more than one option is required, the values can be:

- Added together (do not add the same constant more than once), or
- Combined using the bitwise OR operation (if the programming language supports bit operations).

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 (see Table 60 on page 302).

The following options control the operations that can be performed on an object:

MQOO_INPUT_AS_Q_DEF
: Open queue to get messages using queue-defined default.

  The queue is opened for use with subsequent MQGET calls. The type of access is either shared or exclusive, depending on the value of the *DefInputOpenOption* queue attribute; see "Attributes for local queues and model queues" on page 348 for details.

  This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

MQOO_INPUT_SHARED
: Open queue to get messages with shared access.

  The queue is opened for use with subsequent MQGET calls. The call can succeed if the queue is currently open by this or another application with MQOO_INPUT_SHARED, but fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open with MQOO_INPUT_EXCLUSIVE.

  This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

MQOO_INPUT_EXCLUSIVE
: Open queue to get messages with exclusive access.

  The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open by this or another application for input of any type (MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE).

  This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects that are not queues.

The following notes apply to:

MQOO_INPUT_AS_Q_DEF
MQOO_INPUT_SHARED
MQOO_INPUT_EXCLUSIVE

- Only one of these options can be specified.
- An MQOPEN call with one of these options can succeed even if the *InhibitGet* queue attribute is set to MQQA_GET_INHIBITED

(although subsequent MQGET calls will fail while the attribute is set to this value).

- If the queue is defined as not being shareable (that is, the *Shareability* local-queue attribute has the value MQQA_NOT_SHAREABLE), attempts to open the queue for shared access are treated as attempts to open the queue with exclusive access.
- If an alias queue is opened with one of these options, the test for exclusive use (or for whether another application has exclusive use) is against the base queue to which the alias resolves.
- These options are not valid if *ObjectQMgrName* is the name of a queue manager alias; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

MQOO_BROWSE
Open queue to browse messages.

The queue is opened for use with subsequent MQGET calls with one of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_NEXT
MQGMO_BROWSE_MSG_UNDER_CURSOR

This is allowed even if the queue is currently open for MQOO_INPUT_EXCLUSIVE. An MQOPEN call with the MQOO_BROWSE option establishes a browse cursor, and positions it logically before the first message on the queue; see the *Options* field descibed in “MQGMO – Get-message options” on page 56 for further information.

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects which are not queues. It is also not valid if *ObjectQMgrName* is the name of a queue manager alias; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

MQOO_OUTPUT
Open queue to put messages.

The queue is opened for use with subsequent MQPUT calls.

An MQOPEN call with this option can succeed even if the *InhibitPut* queue attribute is set to MQQA_PUT_INHIBITED (although subsequent MQPUT calls will fail while the attribute is set to this value).

This option is valid for all types of queue, including distribution lists.

MQOO_INQUIRE
Open object to inquire attributes.

The queue, namelist, process definition, or queue manager is opened for use with subsequent MQINQ calls.

This option is valid for all types of object other than distribution lists. It is not valid if *ObjectQMgrName* is the name of a queue manager alias; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

MQOO_SET
Open queue to set attributes.

The queue is opened for use with subsequent MQSET calls.

This option is valid for all types of queue other than distribution lists. It is not valid if *ObjectQMgrName* is the name of a local definition of a remote queue; this is true even if the value of the *RemoteQMgrName* attribute in the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

The following options control the processing of message context:

MQOO_SAVE_ALL_CONTEXT
Save context when message retrieved.

Context information is associated with this queue handle. This information is set from the context of any message retrieved using this handle. For more information on message context, see the *MQSeries Application Programming Guide*.

This context information can be passed to a message that is subsequently put on a queue using the MQPUT or MQPUT1 calls. See the MQPMO_PASS_IDENTITY_CONTEXT and MQPMO_PASS_ALL_CONTEXT options described in “MQPMO – Put message options” on page 173.

Until a message has been successfully retrieved, context cannot be passed to a message being put on a queue.

A message retrieved using one of the MQGMO_BROWSE_★ browse options does **not** have its context information saved (although the context fields in the *MsgDesc* parameter are set after a browse).

This option is valid only for local, alias, and model queues; it is not valid for remote queues, distribution lists, and objects which are not queues. One of the MQOO_INPUT_★ options must be specified.

This option is not supported in the following environments: 16-bit Windows, 32-bit Windows.

MQOO_PASS_IDENTITY_CONTEXT
Allow identity context to be passed.

This allows the MQPMO_PASS_IDENTITY_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity context information from an input queue that was opened with the MQOO_SAVE_ALL_CONTEXT option. For more information on message context, see the *MQSeries Application Programming Guide*.

The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

This option is not supported in the following environments: 16-bit Windows, 32-bit Windows.

MQOO_PASS_ALL_CONTEXT
Allow all context to be passed.

This allows the MQPMO_PASS_ALL_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity and origin context information from an input queue that was opened with the MQOO_SAVE_ALL_CONTEXT option. For more information on message context, see the *MQSeries Application Programming Guide*.

This option implies MQOO_PASS_IDENTITY_CONTEXT, which need not therefore be specified. The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

This option is not supported in the following environments: 16-bit Windows, 32-bit Windows.

MQOO_SET_IDENTITY_CONTEXT
Allow identity context to be set.

This allows the MQPMO_SET_IDENTITY_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity context information contained in the *MsgDesc* parameter specified on the MQPUT or MQPUT1 call. For more information on message context, see the *MQSeries Application Programming Guide*.

This option implies MQOO_PASS_IDENTITY_CONTEXT, which need not therefore be specified. The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

MQOO_SET_ALL_CONTEXT
Allow all context to be set.

This allows the MQPMO_SET_ALL_CONTEXT option to be specified in the *PutMsgOpts* parameter when a message is put on a queue; this gives the message the identity and origin context information contained in the *MsgDesc* parameter specified on the MQPUT or MQPUT1 call. For more information on message context, see the *MQSeries Application Programming Guide*.

This option implies the following options, which need not therefore be specified:

MQOO_PASS_IDENTITY_CONTEXT
MQOO_PASS_ALL_CONTEXT
MQOO_SET_IDENTITY_CONTEXT

The MQOO_OUTPUT option must be specified.

This option is valid for all types of queue, including distribution lists.

The following options control authorization checking, and what happens when the queue manager is quiescing:

MQOO_ALTERNATE_USER_AUTHORITY
Validate with specified user identifier.

This indicates that the *AlternateUserId* field in the *ObjDesc* parameter contains a user identifier that is to be used to validate this MQOPEN call. The call can succeed only if this *AlternateUserId* is authorized to open the object with the specified options, regardless of whether the user identifier under which the application is running is authorized to do so. (This does not apply to any context options specified, however, which are always checked against the user identifier under which the application is running.)

This option is valid for all types of object.

In the following environments, this option is accepted but ignored: 16-bit Windows, 32-bit Windows.

MQOO_FAIL_IF_QUIESCING
Fail if queue manager is quiescing.

This option forces the MQOPEN call to fail if the queue manager is in quiescing state.

On MVS/ESA, for a CICS or IMS application, this option also forces the MQOPEN call to fail if the connection is in quiescing state.

This option is valid for all types of object.

In the following environments, this option is accepted but ignored: 16-bit Windows, 32-bit Windows.

Table 60. Valid MQOPEN options for each queue type

Option	Alias (see note 1)	Local	Model	Remote	Distribution list
MQOO_INPUT_AS_Q_DEF	√	√	√	—	—
MQOO_INPUT_SHARED	√	√	√	—	—
MQOO_INPUT_EXCLUSIVE	√	√	√	—	—
MQOO_BROWSE	√	√	√	—	—
MQOO_OUTPUT	√	√	√	√	√
MQOO_INQUIRE	√	√	√	√ (see note 2)	—
MQOO_SET	√	√	√	√ (see note 2)	—
MQOO_SAVE_ALL_CONTEXT	√	√	√	—	—
MQOO_PASS_IDENTITY_CONTEXT	√	√	√	√	√
MQOO_PASS_ALL_CONTEXT	√	√	√	√	√
MQOO_SET_IDENTITY_CONTEXT	√	√	√	√	√
MQOO_SET_ALL_CONTEXT	√	√	√	√	√
MQOO_ALTERNATE_USER_AUTHORITY	√	√	√	√	√
MQOO_FAIL_IF_QUIESCING	√	√	√	√	√

Notes:

1. The validity of options for aliases depends on the validity of the option for the queue to which the alias resolves.
2. This option is valid only for the local definition of a remote queue.

Hobj (MQHOBJ) – 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 unit of processing that defines the scope of the handle terminates.

The scope of the handle is restricted to the smallest unit of parallel processing within the environment concerned; the handle is not valid outside the unit of parallel processing from which the MQOPEN call was issued:

- On OpenVMS, the scope of the handle is the thread issuing the call.
- On DOS client, the scope of the handle is the system.
- On MVS/ESA, the scope of the handle is:
  - For CICS, the CICS task issuing the call
  - For IMS, the Task Control Block issuing the call, up to the next syncpoint; this excludes any subtasks of the task
  - For MVS batch, the Task Control Block; this excludes any subtasks of the task.
- On OS/2, the scope of the handle is the thread issuing the call.
- On OS/400, the scope of the handle is the job issuing the call.
- On Tandem NSK, the scope of the handle is the thread issuing the call.
- On UNIX systems, the scope of the handle is the thread issuing the call.
- On Windows client and 16-bit Windows, the scope of the handle is the process issuing the call.
- On 32-bit Windows and Windows NT, the scope of the handle is the thread issuing the call.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_ALIAS_BASE_Q_TYPE_ERROR
(2001, X'7D1') Alias base queue not a valid type.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_QUIESCING
(2202, X'89A') Connection quiescing.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_DEF_XMIT_Q_TYPE_ERROR
(2198, X'896') Default transmission queue not local.
MQRC_DEF_XMIT_Q_USAGE_ERROR
(2199, X'897') Default transmission queue usage error.
MQRC_DYNAMIC_Q_NAME_ERROR
(2011, X'7DB') Name of dynamic queue not valid.
MQRC_HANDLE_NOT_AVAILABLE
(2017, X'7E1') No more handles available.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.
MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.
MQRC_NAME_IN_USE
(2201, X'899') Name in use.
MQRC_NAME_NOT_VALID_FOR_TYPE
(2194, X'892') Object name not valid for object type.
MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.
MQRC_OBJECT_ALREADY_EXISTS
(2100, X'834') Object already exists.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_OBJECT_IN_USE
(2042, X'7FA') Object already open with conflicting options.
MQRC_OBJECT_NAME_ERROR
(2152, X'868') Object name not valid.
MQRC_OBJECT_Q_MGR_NAME_ERROR
(2153, X'869') Object queue-manager name not valid.
MQRC_OBJECT_RECORDS_ERROR
(2155, X'86B') Object records not valid.

MQRC_OBJECT_TYPE_ERROR
(2043, X'7FB') Object type not valid.
MQRC_OD_ERROR
(2044, X'7FC') Object descriptor structure not valid.
MQRC_OPTION_NOT_VALID_FOR_TYPE
(2045, X'7FD') Option not valid for object type.
MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_PAGESET_FULL
(2192, X'890') Page set data set full.
MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_Q_TYPE_ERROR
(2057, X'809') Queue type not valid.
MQRC_RECS_PRESENT_ERROR
(2154, X'86A') Number of records present not valid.
MQRC_REMOTE_Q_NAME_ERROR
(2184, X'888') Remote queue name not valid.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_RESPONSE_RECORDS_ERROR
(2156, X'86C') Response records not valid.
MQRC_SECURITY_ERROR
(2063, X'80F') Security error occurred.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.
MQRC_UNKNOWN_ALIAS_BASE_Q
(2082, X'822') Unknown alias base queue.
MQRC_UNKNOWN_DEF_XMIT_Q
(2197, X'895') Unknown default transmission queue.
MQRC_UNKNOWN_OBJECT_NAME
(2085, X'825') Unknown object name.
MQRC_UNKNOWN_OBJECT_Q_MGR
(2086, X'826') Unknown object queue manager.
MQRC_UNKNOWN_REMOTE_Q_MGR
(2087, X'827') Unknown remote queue manager.
MQRC_UNKNOWN_XMIT_Q
(2196, X'894') Unknown transmission queue.
MQRC_XMIT_Q_TYPE_ERROR
(2091, X'82B') Transmission queue not local.

MQRC_XMIT_Q_USAGE_ERROR
(2092, X'82C') Transmission queue with wrong usage.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. The object opened is one of the following:
   - A queue, in order to:
     – Get or browse messages (using the MQGET call)
     – Put messages (using the MQPUT call)
     – Inquire about the attributes of the queue (using the MQINQ call)
     – Set the attributes of the queue (using the MQSET call)

     If the queue named is a model queue, a dynamic local queue is created. See the *ObjDesc* parameter described in “MQOPEN – Open object” on page 297.

     A distribution list is a special type of queue object that contains a list of queues. It can be opened to put messages, but not to get or browse messages, or to inquire or set attributes.
   - A namelist, in order to:
     – Inquire about the names of the queues in the list (using the MQINQ call).

     Namelists are supported only on MVS/ESA.
   - A process definition, in order to:
     – Inquire about the process attributes (using the MQINQ call).
   - The queue manager, in order to:
     – Inquire about the attributes of the local queue manager (using the MQINQ call).
2. It is valid for an application to open the same object more than once. A different object handle is returned for each open. Each handle that is returned can be used for the functions for which the corresponding open was performed.
3. All name resolution within the local queue manager 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 the name of a base queue
   - Resolution of the name of a local definition of a remote queue to the remote queue-manager name and the name by which that queue is known at the remote queue manager
   - Resolution of the remote queue-manager name to the name of a transmission queue

   However, be aware that subsequent MQINQ or MQSET 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. Name

resolution checking is still carried out, however, regardless of what is specified for the *Options* parameter on the corresponding MQOPEN.

4. The attributes of an object can change while an application has the object open. In many cases, the application does not notice this, but for certain attributes the queue manager marks the handle as no longer valid. These are:

   - Any attribute that affects the name resolution of the object (see Usage note 3 on page 306), regardless of the open options used. This includes the following:

     – A change to the *BaseQName* of an alias queue that is open.

     – With one exception, any change that causes a currently-open handle for a remote queue to resolve to a different transmission queue, or to fail to resolve to one at all. For example, a change to the *XmitQName* attribute of the local definition of a remote queue, whether the definition is being used for a queue, or for a queue-manager alias.

       The exception is the creation of a new transmission queue. A handle that would have resolved to this queue, had it been present when the handle was opened, but instead resolved to the default transmission queue, is not made invalid.

     – A change to the *DefXmitQName* queue-manager attribute. In this case all open handles that resolved to the previously-named queue (that resolved to it only because it was the default transmission queue) are marked as invalid. Handles that resolved to this queue for other reasons are not affected.

       On MVS/ESA, the *DefXmitQName* attribute is not supported.

     – The *RemoteQName* or *RemoteQMgrName* remote queue attributes, for any handle that is open for this queue, or for a queue which resolves through this definition as a queue-manager alias.

   - The *Shareability* local queue attribute, if there are two or more handles that are currently providing MQOO_INPUT_SHARED access for this queue, or for a queue that resolves to this queue. If this is the case, *all* handles that are open for this queue, or for a queue that resolves to this queue, are marked as invalid, regardless of the open options.

     On MVS/ESA, the handles described above are marked as invalid if one or more handles is currently providing MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE access to the queue.

   - The *Usage* local queue attribute, for all handles that are open for this queue, or for a queue that resolves to this queue, regardless of the open options.

   When a handle is marked as invalid, all subsequent calls (other than MQCLOSE) using this handle fail with reason code MQRC_OBJECT_CHANGED; the application should issue an MQCLOSE call (using the original handle) and then reopen the queue. Any uncommitted updates against the old handle from previous successful calls can still be committed or backed out, as required by the application logic.

   If changing an attribute will cause this to happen, a special “force” version of the command must be used.

5. The queue manager performs security checks when an MQOPEN call is issued, to verify that the user identifier under which the application is running

has the appropriate level of authority before access is permitted. The authority check is made on the name of the object being opened, and not on the name, or names, resulting after a name has been resolved.

On MVS/ESA, the queue manager performs security checks only if security is enabled. For more information on security checking, see the *MQSeries for MVS/ESA System Management Guide*.

6. If the object being opened is a model queue, the queue manager performs a full security check against both the name of the model queue and the name of the dynamic queue that is created. If the resulting dynamic queue is subsequently opened explicitly, a further resource security check is performed against the name of the dynamic queue.

7. A remote queue can be specified in one of two ways in the *ObjDesc* parameter of this call (see the *ObjectName* and *ObjectQMgrName* fields described in "MQOD – Object descriptor" on page 160):

   - By specifying for *ObjectName* the name of a local definition of the remote queue. In this case, *ObjectQMgrName* refers to the local queue manager, and can be specified as blanks or (in the C programming language) a null string.

     The security validation performed by the local queue manager verifies that the application is authorized to open the local definition of the remote queue.

   - By specifying for *ObjectName* the name of the remote queue as known to the remote queue manager. In this case, *ObjectQMgrName* is the name of the remote queue manager.

     The security validation performed by the local queue manager verifies that the application is authorized to send messages to the transmission queue resulting from the name resolution process.

   In either case:

   - No messages are sent by the local queue manager to the remote queue manager in order to check that the application is authorized to put messages on the queue.
   - When a message arrives at the remote queue manager, the remote queue manager may reject it because the user originating the message is not authorized.

8. The following notes apply to the use of distribution lists.

   Distribution lists are supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

   a. Fields in the MQOD structure must be set as follows when opening a distribution list:

      - *Version* must be MQOD_VERSION_2.
      - *ObjectType* must be MQOT_Q.
      - *ObjectName* must be blank or the null string.
      - *ObjectQMgrName* must be blank or the null string.
      - *RecsPresent* must be greater than zero.
      - One of *ObjectRecOffset* and *ObjectRecPtr* must be zero and the other nonzero.

- No more than one of *ResponseRecOffset* and *ResponseRecPtr* can be nonzero.
- There must be *RecsPresent* object records, addressed by either *ObjectRecOffset* or *ObjectRecPtr*. The object records must be set to the names of the destination queues to be opened.
- If one of *ResponseRecOffset* and *ResponseRecPtr* is nonzero, there must be *RecsPresent* response records present. They are set by the queue manager if the call completes with reason code MQRC_MULTIPLE_REASONS.

A version-2 MQOD can also be used to open a single queue that is not in a distribution list, by ensuring that *RecsPresent* is zero.

b. Only the following open options are valid in the *Options* parameter:

MQOO_OUTPUT
MQOO_PASS_*_CONTEXT
MQOO_SET_*_CONTEXT
MQOO_ALTERNATE_USER_AUTHORITY
MQOO_FAIL_IF_QUIESCING

c. The destination queues in the distribution list can be local, alias, or remote queues, but they cannot be model queues. If a model queue is specified, that queue fails to open, with reason code MQRC_Q_TYPE_ERROR. However, this does not prevent other queues in the list being opened successfully.

d. The completion code and reason code parameters are set as follows:

- If the open operations for the queues in the distribution list all succeed or fail in the same way, the completion code and reason code parameters are set to describe the common result. The MQRR response records (if provided by the application) are not set in this case.

  For example, if every open succeeds, the completion code and reason code are set to MQCC_OK and MQRC_NONE respectively; if every open fails because none of the queues exists, the parameters are set to MQCC_FAILED and MQRC_UNKNOWN_OBJECT_NAME.

- If the open operations for the queues in the distribution list do not all succeed or fail in the same way:
  - The completion code parameter is set to MQCC_WARNING if at least one open succeeded, and to MQCC_FAILED if all failed.
  - The reason code parameter is set to MQRC_MULTIPLE_REASONS.
  - The response records (if provided by the application) are set to the individual completion codes and reason codes for the queues in the distribution list.

e. When a distribution list has been opened successfully, the handle *Hobj* returned by the call can be used on subsequent MQPUT calls to put messages to queues in the distribution list, and on an MQCLOSE call to relinquish access to the distribution list. The only valid close option for a distribution list is MQCO_NONE.

The MQPUT1 call can also be used to put a message to a distribution list; the MQOD structure defining the queues in the list is specified as a parameter on that call.

f. Each successfully-opened destination in the distribution list counts as a separate handle when checking whether the application has exceeded the permitted maximum number of handles (see the *MaxHandles* queue-manager attribute). This is true even when two or more of the destinations in the distribution list actually resolve to the same physical queue.

   In a similar fashion, each destination that is opened successfully may have the value of its *OpenOutputCount* attribute incremented by one.

g. Any change to the queue definitions that would have caused a handle to become invalid had the queues been opened individually (for example, a change in the resolution path), does not cause the distribution-list handle to become invalid. However, it does result in a failure for that particular queue when the distribution-list handle is used on a subsequent MQPUT call.

h. It is valid for a distribution list to contain only one destination.

9. An MQOPEN call with the MQOO_BROWSE option establishes a browse cursor, for use with 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 by using the MQGMO_MSG_UNDER_CURSOR option.

   Multiple browse cursors can be active for a single application by issuing several MQOPEN requests for the same queue.

10. On OS/400, the first MQOPEN call performs an implicit MQCONN function, if MQCONN has not already been issued.

11. Applications started by a trigger monitor are passed the name of the queue that is associated with the application when the application is started. This queue name can be specified in the *ObjDesc* parameter to open the queue. See the description of the MQTMC2 structure for further details.

C language invocation

```
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 */
```

COBOL language invocation

```
     CALL 'MQOPEN' USING HCONN, OBJDESC, OPTIONS, HOBJ,
                         COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN     PIC S9(9) BINARY.
**   Object descriptor
 01  OBJDESC.
     COPY CMQODV.
**   Options that control the action of MQOPEN
 01  OPTIONS   PIC S9(9) BINARY.
**   Object handle
 01  HOBJ      PIC S9(9) BINARY.
**   Completion code
 01  COMPCODE  PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON    PIC S9(9) BINARY.
```

PL/I invocation (AIX, MVS/ESA, OS/2 and Windows NT)

```
call MQOPEN (Hconn, ObjDesc, Options, Hobj, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn     fixed bin(31);  /* Connection handle */
dcl ObjDesc   like MQOD;      /* Object descriptor */
dcl Options   fixed bin(31);  /* Options that control the action of
                                 MQOPEN */
dcl Hobj      fixed bin(31);  /* Object handle */
dcl CompCode  fixed bin(31);  /* Completion code */
dcl Reason    fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
         CALL MQOPEN,(HCONN,OBJDESC,OPTIONS,HOBJ,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
OBJDESC             CMQODA              Object descriptor
OPTIONS             DS      F           Options that control the action
*                                       of MQOPEN
HOBJ                DS      F           Object handle
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn;
STRUCT    .EXT ObjDesc(MQOD^Def);
INT(32)      Options; INT(32) .EXT Hobj;
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL  MQOPEN(HConn, ObjDesc, Options, HObj, CC, Reason);
```

MQPUT – Put message

The MQPUT call puts a message on a queue or distribution list. The queue or distribution list must already be open.

```
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer,
       CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

Hobj (MQHOBJ) – input
Object handle.

This handle represents the queue to which the message is added. The value of *Hobj* was returned by a previous MQOPEN call that specified the MQOO_OUTPUT option.

MsgDesc (MQMD) – input/output
Message descriptor.

This structure describes the attributes of the message being sent, and receives information about the message after the put request is complete. See “MQMD – Message descriptor” on page 98 for details.

If the application provides a version-1 MQMD, the message data can be prefixed with an MQMDE structure in order to specify values for the fields that exist in the version-2 MQMD but not the version-1. The *Format* field in the MQMD must be set to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. See “MQMDE – Message descriptor extension” on page 153 for more details.

PutMsgOpts (MQPMO) – input/output
Options that control the action of MQPUT.

See “MQPMO – Put message options” on page 173 for details.

BufferLength (MQLONG) – input
Length of the message in *Buffer*.

Zero is valid, and indicates that the message contains no application data.

If the destination is a local queue, or resolves to a local queue, the upper limit for *BufferLength* depends on whether:

- The local queue manager supports segmentation.

- The sending application specifies the flag that allows the queue manager to segment the message.

  This flag is MQMF_SEGMENTATION_ALLOWED, and can be specified either in a version-2 MQMD, or in an MQMDE used with a version-1 MQMD.

If both of these conditions are satisfied, there is no specific upper limit on the length of the message that can be put. Resource constraints imposed by the operating system or by the environment in which the application is running will impose some upper limit, but that limit can be greater than either the queue's *MaxMsgLength* attribute or queue-manager's *MaxMsgLength* attribute.

If one or both of the above conditions is not satisfied, *BufferLength* cannot exceed the smaller of the queue's *MaxMsgLength* attribute and queue-manager's *MaxMsgLength* attribute.

If the destination is a remote queue, or resolves to a remote queue, the same conditions apply, *but at each queue manager through which the message must pass in order to reach the destination queue*; in particular:

1. The local transmission queue used to store the message temporarily at the local queue manager
2. Intermediate transmission queues (if any) used to store the message at queue managers on the route between the local and destination queue managers
3. The destination queue at the destination queue manager

The longest message that can be put is therefore governed by the most-restrictive of these queues and queue managers.

When a message is on a transmission queue, additional information resides with the message data, and this reduces the amount of application data that can be carried. In this situation it is recommended that MQ_MSG_HEADER_LENGTH bytes be subtracted from the *MaxMsgLength* values of the transmission queues when determining the limit for *BufferLength*.

Note: Only failure to comply with condition 1 can be diagnosed synchronously (with reason code MQRC_MSG_TOO_BIG_FOR_Q or MQRC_MSG_TOO_BIG_FOR_Q_MGR) when the message is put. If conditions 2 or 3 are not satisfied, the message is redirected to a dead-letter (undelivered-message) queue, either at an intermediate queue manager or at the destination queue manager. If this happens, a report message is generated if one was requested by the sender.

Buffer (MQBYTE×*BufferLength*) – input
Message data.

This is a buffer containing the application data to be sent.

If *Buffer* contains character and/or numeric data, the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter should be set to the values appropriate to the data; this will enable the receiver of the message to convert the data (if necessary) to the character set and encoding used by the receiver.

Note: All of the other parameters on the MQPUT call must be in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE, respectively).

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 the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler C can be null.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.
MQRC_PRIORITY_EXCEEDS_MAXIMUM
(2049, X'801') Message Priority exceeds maximum value supported.
MQRC_UNKNOWN_REPORT_OPTION
(2104, X'838') Report option(s) in message descriptor not recognized.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_BACKED_OUT
(2003, X'7D3') Unit of work encountered fatal error or backed out.
MQRC_BUFFER_ERROR
(2004, X'7D4') Buffer parameter not valid.
MQRC_BUFFER_LENGTH_ERROR
(2005, X'7D5') Buffer length parameter not valid.

MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_COD_NOT_VALID_FOR_XCF_Q
(2106, X'83A') COD report option not valid for XCF queue.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_QUIESCING
(2202, X'89A') Connection quiescing.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_CONTEXT_HANDLE_ERROR
(2097, X'831') Queue handle referred to does not save context.
MQRC_CONTEXT_NOT_AVAILABLE
(2098, X'832') Context not available for queue handle referred to.
MQRC_DH_ERROR
(2135, X'857') Distribution header structure not valid.
MQRC_EXPIRY_ERROR
(2013, X'7DD') Expiry time not valid.
MQRC_FEEDBACK_ERROR
(2014, X'7DE') Feedback code not valid.
MQRC_GROUP_ID_ERROR
(2258, X'8D2') Group identifier not valid.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.
MQRC_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.
MQRC_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.
MQRC_INCONSISTENT_PERSISTENCE
(2185, X'889') Inconsistent persistence specification.
MQRC_INCONSISTENT_UOW
(2245, X'8C5') Inconsistent unit-of-work specification.
MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.
MQRC_MDE_ERROR
(2248, X'8C8') Message descriptor extension not valid.
MQRC_MISSING_REPLY_TO_Q
(2027, X'7EB') Missing reply-to queue.
MQRC_MSG_FLAGS_ERROR
(2249, X'8C9') Message flags not valid.
MQRC_MSG_SEQ_NUMBER_ERROR
(2250, X'8CA') Message sequence number not valid.
MQRC_MSG_TOO_BIG_FOR_Q
(2030, X'7EE') Message length greater than maximum for queue.
MQRC_MSG_TYPE_ERROR
(2029, X'7ED') Message type in message descriptor not valid.
MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.

MQRC_NOT_OPEN_FOR_OUTPUT
(2039, X'7F7') Queue not open for output.
MQRC_NOT_OPEN_FOR_PASS_ALL
(2093, X'82D') Queue not open for pass all context.
MQRC_NOT_OPEN_FOR_PASS_IDENT
(2094, X'82E') Queue not open for pass identity context.
MQRC_NOT_OPEN_FOR_SET_ALL
(2095, X'82F') Queue not open for set all context.
MQRC_NOT_OPEN_FOR_SET_IDENT
(2096, X'830') Queue not open for set identity context.
MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_OFFSET_ERROR
(2251, X'8CB') Message segment offset not valid.
MQRC_OPEN_FAILED
(2137, X'859') Queue not opened successfully.
MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_ORIGINAL_LENGTH_ERROR
(2252, X'8CC') Original length not valid.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_PAGESET_FULL
(2192, X'890') Page set data set full.
MQRC_PERSISTENCE_ERROR
(2047, X'7FF') Persistence not valid.
MQRC_PERSISTENT_NOT_ALLOWED
(2048, X'800') Message on a temporary dynamic queue cannot be persistent.
MQRC_PMO_ERROR
(2173, X'87D') Put-message options structure not valid.
MQRC_PMO_RECORD_FLAGS_ERROR
(2158, X'86E') Put message record flags not valid.
MQRC_PRIORITY_ERROR
(2050, X'802') Message priority not valid.
MQRC_PUT_INHIBITED
(2051, X'803') Put calls inhibited for the queue.
MQRC_PUT_MSG_RECORDS_ERROR
(2159, X'86F') Put message records not valid.
MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.
MQRC_Q_FULL
(2053, X'805') Queue already contains maximum number of messages.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.

MQRC_Q_SPACE_NOT_AVAILABLE
(2056, X'808') No space available on disk for queue.
MQRC_RECS_PRESENT_ERROR
(2154, X'86A') Number of records present not valid.
MQRC_REPORT_OPTIONS_ERROR
(2061, X'80D') Report options in message descriptor not valid.
MQRC_RESPONSE_RECORDS_ERROR
(2156, X'86C') Response records not valid.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SEGMENT_LENGTH_ZERO
(2253, X'8CD') Length of data in message segment is zero.
MQRC_STORAGE_CLASS_ERROR
(2105, X'839') Storage class error.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_SYNCPOINT_LIMIT_REACHED
(2024, X'7E8') No more messages can be handled within current unit of work.
MQRC_SYNCPOINT_NOT_AVAILABLE
(2072, X'818') Syncpoint support not available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.
MQRC_UOW_NOT_AVAILABLE
(2255, X'8CF') Unit of work not available for the queue manager to use.
MQRC_WRONG_MD_VERSION
(2257, X'8D1') Wrong version of MQMD supplied.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. Both the MQPUT and MQPUT1 calls can be used to put messages on a queue; which call to use depends on the circumstances:
   - The MQPUT call should be used when multiple messages are to be placed on the *same* queue.

     An MQOPEN call specifying the MQOO_OUTPUT option is issued first, followed by one or more MQPUT requests to add messages to the queue; finally the queue is closed with an MQCLOSE call. This gives better performance than repeated use of the MQPUT1 call.
   - The MQPUT1 call should be used when only *one* message is to be put on a queue.

     This call encapsulates the MQOPEN, MQPUT, and MQCLOSE calls into a single call, thereby minimizing the number of calls that must be issued.
2. The following notes apply to the use of distribution lists.

   Distribution lists are supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

a. Messages can be put to a distribution list using either a version-1 or a version-2 MQPMO. If a version-1 MQPMO is used (or a version-2 MQPMO with *RecsPresent* equal to zero), no put message records or response records can be provided by the application. This means that it will not be possible to identify the queues which encounter errors, if the message is sent successfully to some queues in the distribution list and not others.

   If put message records or response records are provided by the application, the *Version* field must be set to MQPMO_VERSION_2.

   A version-2 MQPMO can also be used to send messages to a single queue that is not in a distribution list, by ensuring that *RecsPresent* is zero.

b. The completion code and reason code parameters are set as follows:

   - If the puts to the queues in the distribution list all succeed or fail in the same way, the completion code and reason code parameters are set to describe the common result. The MQRR response records (if provided by the application) are not set in this case.

     For example, if every put succeeds, the completion code and reason code are set to MQCC_OK and MQRC_NONE respectively; if every put fails because all of the queues are inhibited for puts, the parameters are set to MQCC_FAILED and MQRC_PUT_INHIBITED.

   - If the puts to the queues in the distribution list do not all succeed or fail in the same way:

     – The completion code parameter is set to MQCC_WARNING if at least one put succeeded, and to MQCC_FAILED if all failed.
     – The reason code parameter is set to MQRC_MULTIPLE_REASONS.
     – The response records (if provided by the application) are set to the individual completion codes and reason codes for the queues in the distribution list.

     If the put to a destination fails because the open for that destination failed, the fields in the response record are set to MQCC_FAILED and MQRC_OPEN_FAILED; that destination is included in *InvalidDestCount*.

c. If a destination in the distribution list resolves to a local queue, the message is placed on that queue in normal form (that is, not as a distribution-list message). If more than one destination resolves to the same local queue, one message is placed on the queue for each such destination.

   If a destination in the distribution list resolves to a remote queue, a message is placed on the appropriate transmission queue. Where several destinations resolve to the same transmission queue, a single distribution-list message containing those destinations may be placed on the transmission queue, even if those destinations were not adjacent in the list of destinations provided by the application. However, this can be done only if the transmission queue supports distribution-list messages (see the *DistLists* queue attribute described in “Attributes for local queues and model queues” on page 348).

   If the transmission queue does not support distribution lists, one copy of the message in normal form is placed on the transmission queue for each destination that uses that transmission queue.

If a distribution list with the application message data is too big for a transmission queue, the distribution list message is split up into smaller distribution-list messages, each containing fewer destinations. If the application message data only just fits on the queue, distribution-list messages cannot be used at all, and the queue manager generates one copy of the message in normal form for each destination that uses that transmission queue.

If different destinations have different message priority or message persistence (this can occur when the application specifies MQPRI_PRIORITY_AS_Q_DEF or MQPER_PERSISTENCE_AS_Q_DEF), the messages are not held in the same distribution-list message. Instead, the queue manager generates as many distribution-list messages as are necessary to accommodate the differing priority and persistence values.

d. A put to a distribution list may result in:

- A single distribution-list message, or
- A number of smaller distribution-list messages, or
- A mixture of distribution list messages and normal messages, or
- Normal messages only.

Which of the above occurs depends on whether:

- The destinations in the list are local, remote, or a mixture.
- The destinations have the same message priority and message persistence.
- The transmission queues can hold distribution-list messages.
- The transmission queues' maximum message lengths are large enough to accommodate the message in distribution-list form.

However, regardless of which of the above occurs, each *physical* message resulting (that is, each normal message or distribution-list message resulting from the put) counts as only *one* message when:

- Checking whether the application has exceeded the permitted maximum number of messages in a unit of work (see the *MaxUncommittedMsgs* queue-manager attribute).
- Checking whether the triggering conditions are satisfied.
- Incrementing queue depths and checking whether the queues' maximum queue depth would be exceeded.

e. Any change to the queue definitions that would have caused a handle to become invalid had the queues been opened individually (for example, a change in the resolution path), does not cause the distribution-list handle to become invalid. However, it does result in a failure for that particular queue when the distribution-list handle is used on a subsequent MQPUT call.

3. If a message is put with one or more MQ header structures at the beginning of the application message data, the queue manager performs certain checks on the header structures to verify that they are valid. If the queue manager detects an error, the call fails with an appropriate reason code. The checks performed vary according to the particular structures that are present. In addition, the checks are performed only if a version-2 or later MQMD is used on the MQPUT or MQPUT1 call; the checks are not performed if a version-1 MQMD is used, even if an MQMDE is present at the start of the application message data.

The following MQ header structures are validated completely by the queue manager: MQDH, MQMDE.

For other MQ header structures, the queue manager performs some validation, but does not check every field. Structures that are not supported by the local queue manager, and structures following the first MQDLH in the message, are not validated.

In addition to general checks on the fields in MQ structures, the following conditions must be satisfied:

- An MQ structure must not be split over two or more segments – the structure must be entirely contained within one segment.
- The sum of the lengths of the structures in a PCF message must equal the length specified by the *BufferLength* parameter on the MQPUT or MQPUT1 call. A PCF message is a message that has one of the following format names:

  MQFMT_ADMIN
  MQFMT_EVENT
  MQFMT_PCF

- MQ structures must not be truncated, except in the following situations where truncated structures are permitted:
  - Messages which are report messages.
  - PCF messages.
  - Messages containing an MQDLH structure. (Structures *following* the first MQDLH can be truncated; structures preceding the MQDLH cannot.)

4. On Tandem NSK, if the MQPUT call is issued outside a Tandem TMF transaction *without* the MQPMO_NO_SYNCPOINT option, the reason code MQRC_UNIT_OF_WORK_NOT_STARTED is returned.

C language invocation

```
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 of the message in Buffer */
MQBYTE   Buffer[n];     /* Message data */
MQLONG   CompCode;      /* Completion code */
MQLONG   Reason;        /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQPUT' USING HCONN, HOBJ, MSGDESC, PUTMSGOPTS,
                        BUFFERLENGTH, BUFFER, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN          PIC S9(9) BINARY.
**   Object handle
 01  HOBJ           PIC S9(9) BINARY.
**   Message descriptor
 01  MSGDESC.
     COPY CMQMDV.
**   Options that control the action of MQPUT
 01  PUTMSGOPTS.
     COPY CMQPMOV.
**   Length of the message in Buffer
 01  BUFFERLENGTH  PIC S9(9) BINARY.
**   Message data
 01  BUFFER        PIC X(n).
**   Completion code
 01  COMPCODE      PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON        PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength, Buffer,
            CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn         fixed bin(31);  /* Connection handle */
dcl Hobj          fixed bin(31);  /* Object handle */
dcl MsgDesc       like MQMD;      /* Message descriptor */
dcl PutMsgOpts    like MQPMO;     /* Options that control the action of
                                     MQPUT */
dcl BufferLength  fixed bin(31);  /* Length of the message in Buffer */
dcl Buffer        char(n);        /* Message data */
dcl CompCode      fixed bin(31);  /* Completion code */
dcl Reason        fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
      CALL MQPUT,(HCONN,HOBJ,MSGDESC,PUTMSGOPTS,BUFFERLENGTH,BUFFER,     X
            COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN             DS      F           Connection handle
HOBJ              DS      F           Object handle
MSGDESC           CMQMDA              Message descriptor
PUTMSGOPTS        CMQPMOA             Options that control the action
*                                     of MQPUT
BUFFERLENGTH      DS      F           Length of the message in Buffer
BUFFER            DS      CL(n)       Message data
COMPCODE          DS      F           Completion code
REASON            DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn;
INT(32)   .EXT Hobj;
STRUCT    .EXT MsgDesc(MQMD^Def);
STRUCT    .EXT PutMsgOpt(MQPMO^Def);
INT(32)   .EXT BufferLen
STRING    .EXT Buffer[0:BUFFER^SIZE]
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL MQPUT(HConn, HObj, MsgDesc, PutMsgOpt, BufferLen, Buffer,
                          CC, Reason);
```

MQPUT1 – Put one message

The MQPUT1 call puts one message on a queue. The queue need not be open.

```
MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength, Buffer,
        CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

ObjDesc (MQOD) – input
Object descriptor.

This is a structure which identifies the queue to which the message is added. See “MQOD – Object descriptor” on page 160 for details.

The application must be authorized to open the queue for output. The queue must **not** be a model queue.

MsgDesc (MQMD) – 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 – Message descriptor” on page 98 for details.

If the application provides a version-1 MQMD, the message data can be prefixed with an MQMDE structure in order to specify values for the fields that exist in the version-2 MQMD but not the version-1. The *Format* field in the MQMD must be set to MQFMT_MD_EXTENSION to indicate that an MQMDE is present. See “MQMDE – Message descriptor extension” on page 153 for more details.

PutMsgOpts (MQPMO) – input/output
Options that control the action of MQPUT1.

See “MQPMO – Put message options” on page 173 for details.

BufferLength (MQLONG) – input
Length of the message in *Buffer*.

Zero is valid, and indicates that the message contains no application data. The upper limit depends on various factors; see the description of the *BufferLength* parameter of the MQPUT call for further details.

Buffer (MQBYTE×*BufferLength*) – input
Message data.

This is a buffer containing the application message data to be sent.

If *Buffer* contains character and/or numeric data, the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter should be set to the values appropriate to the data; this will enable the receiver of the message to convert the data (if necessary) to the character set and encoding used by the receiver.

Note: All of the other parameters on the MQPUT1 call must be in the character set and encoding of the local queue manager (given by the *CodedCharSetId* queue-manager attribute and MQENC_NATIVE, respectively).

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 the *BufferLength* parameter is zero, *Buffer* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler can be null.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.
MQRC_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.
MQRC_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.
MQRC_PRIORITY_EXCEEDS_MAXIMUM
(2049, X'801') Message Priority exceeds maximum value supported.
MQRC_UNKNOWN_REPORT_OPTION
(2104, X'838') Report option(s) in message descriptor not recognized.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.
MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_ALIAS_BASE_Q_TYPE_ERROR
(2001, X'7D1') Alias base queue not a valid type.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_BACKED_OUT
(2003, X'7D3') Unit of work encountered fatal error or backed out.
MQRC_BUFFER_ERROR
(2004, X'7D4') Buffer parameter not valid.
MQRC_BUFFER_LENGTH_ERROR
(2005, X'7D5') Buffer length parameter not valid.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_COD_NOT_VALID_FOR_XCF_Q
(2106, X'83A') COD report option not valid for XCF queue.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_QUIESCING
(2202, X'89A') Connection quiescing.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_CONTEXT_HANDLE_ERROR
(2097, X'831') Queue handle referred to does not save context.
MQRC_CONTEXT_NOT_AVAILABLE
(2098, X'832') Context not available for queue handle referred to.
MQRC_DEF_XMIT_Q_TYPE_ERROR
(2198, X'896') Default transmission queue not local.
MQRC_DEF_XMIT_Q_USAGE_ERROR
(2199, X'897') Default transmission queue usage error.
MQRC_DH_ERROR
(2135, X'857') Distribution header structure not valid.
MQRC_EXPIRY_ERROR
(2013, X'7DD') Expiry time not valid.
MQRC_FEEDBACK_ERROR
(2014, X'7DE') Feedback code not valid.
MQRC_GROUP_ID_ERROR
(2258, X'8D2') Group identifier not valid.
MQRC_HANDLE_NOT_AVAILABLE
(2017, X'7E1') No more handles available.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.
MQRC_MDE_ERROR
(2248, X'8C8') Message descriptor extension not valid.

MQRC_MISSING_REPLY_TO_Q
(2027, X'7EB') Missing reply-to queue.
MQRC_MSG_FLAGS_ERROR
(2249, X'8C9') Message flags not valid.
MQRC_MSG_SEQ_NUMBER_ERROR
(2250, X'8CA') Message sequence number not valid.
MQRC_MSG_TOO_BIG_FOR_Q
(2030, X'7EE') Message length greater than maximum for queue.
MQRC_MSG_TYPE_ERROR
(2029, X'7ED') Message type in message descriptor not valid.
MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.
MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_OBJECT_IN_USE
(2042, X'7FA') Object already open with conflicting options.
MQRC_OBJECT_NAME_ERROR
(2152, X'868') Object name not valid.
MQRC_OBJECT_Q_MGR_NAME_ERROR
(2153, X'869') Object queue-manager name not valid.
MQRC_OBJECT_RECORDS_ERROR
(2155, X'86B') Object records not valid.
MQRC_OBJECT_TYPE_ERROR
(2043, X'7FB') Object type not valid.
MQRC_OD_ERROR
(2044, X'7FC') Object descriptor structure not valid.
MQRC_OFFSET_ERROR
(2251, X'8CB') Message segment offset not valid.
MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_ORIGINAL_LENGTH_ERROR
(2252, X'8CC') Original length not valid.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_PAGESET_FULL
(2192, X'890') Page set data set full.
MQRC_PERSISTENCE_ERROR
(2047, X'7FF') Persistence not valid.
MQRC_PERSISTENT_NOT_ALLOWED
(2048, X'800') Message on a temporary dynamic queue cannot be persistent.
MQRC_PMO_ERROR
(2173, X'87D') Put-message options structure not valid.
MQRC_PMO_RECORD_FLAGS_ERROR
(2158, X'86E') Put message record flags not valid.
MQRC_PRIORITY_ERROR
(2050, X'802') Message priority not valid.
MQRC_PUT_INHIBITED
(2051, X'803') Put calls inhibited for the queue.
MQRC_PUT_MSG_RECORDS_ERROR
(2159, X'86F') Put message records not valid.

MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.
MQRC_Q_FULL
(2053, X'805') Queue already contains maximum number of messages.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_Q_SPACE_NOT_AVAILABLE
(2056, X'808') No space available on disk for queue.
MQRC_Q_TYPE_ERROR
(2057, X'809') Queue type not valid.
MQRC_RECS_PRESENT_ERROR
(2154, X'86A') Number of records present not valid.
MQRC_REMOTE_Q_NAME_ERROR
(2184, X'888') Remote queue name not valid.
MQRC_REPORT_OPTIONS_ERROR
(2061, X'80D') Report options in message descriptor not valid.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_RESPONSE_RECORDS_ERROR
(2156, X'86C') Response records not valid.
MQRC_SECURITY_ERROR
(2063, X'80F') Security error occurred.
MQRC_SEGMENT_LENGTH_ZERO
(2253, X'8CD') Length of data in message segment is zero.
MQRC_STORAGE_CLASS_ERROR
(2105, X'839') Storage class error.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_SYNCPOINT_LIMIT_REACHED
(2024, X'7E8') No more messages can be handled within current unit of work.
MQRC_SYNCPOINT_NOT_AVAILABLE
(2072, X'818') Syncpoint support not available.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.
MQRC_UNKNOWN_ALIAS_BASE_Q
(2082, X'822') Unknown alias base queue.
MQRC_UNKNOWN_DEF_XMIT_Q
(2197, X'895') Unknown default transmission queue.
MQRC_UNKNOWN_OBJECT_NAME
(2085, X'825') Unknown object name.
MQRC_UNKNOWN_OBJECT_Q_MGR
(2086, X'826') Unknown object queue manager.
MQRC_UNKNOWN_REMOTE_Q_MGR
(2087, X'827') Unknown remote queue manager.

MQRC_UNKNOWN_XMIT_Q
(2196, X'894') Unknown transmission queue.
MQRC_UOW_NOT_AVAILABLE
(2255, X'8CF') Unit of work not available for the queue manager to use.
MQRC_WRONG_MD_VERSION
(2257, X'8D1') Wrong version of MQMD supplied.
MQRC_XMIT_Q_TYPE_ERROR
(2091, X'82B') Transmission queue not local.
MQRC_XMIT_Q_USAGE_ERROR
(2092, X'82C') Transmission queue with wrong usage.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. Both the MQPUT and MQPUT1 calls can be used to put messages on a queue; which call to use depends on the circumstances:

   - The MQPUT call should be used when multiple messages are to be placed on the *same* queue.

     An MQOPEN call specifying the MQOO_OUTPUT option is issued first, followed by one or more MQPUT requests to add messages to the queue; finally the queue is closed with an MQCLOSE call. This gives better performance than repeated use of the MQPUT1 call.

   - The MQPUT1 call should be used when only *one* message is to be put on a queue.

     This call encapsulates the MQOPEN, MQPUT, and MQCLOSE calls into a single call, thereby minimizing the number of calls that must be issued.

2. The MQPUT1 call can be used to put messages to distribution lists. For general information about this, see usage note 8 on page 308 for the MQOPEN call, and usage note 2 on page 318 for the MQPUT call.

   Distribution lists are supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

   The following differences apply when using the MQPUT1 call:

   a. If MQRR response records are provided by the application, they must be provided using the MQOD structure; they cannot be provided using the MQPMO structure.

   b. The reason code MQRC_OPEN_FAILED is never returned by MQPUT1 in the response records; if a queue fails to open, the response record for that queue contains the actual reason code resulting from the open operation.

      If an open operation for a queue succeeds with a completion code of MQCC_WARNING, the completion code and reason code in the response record for that queue are replaced by the completion and reason codes resulting from the put operation.

      As with the MQOPEN and MQPUT calls, the queue manager sets the response records (if provided) only when the outcome of the call is not the same for all queues in the distribution list; this is indicated by the call completing with reason code MQRC_MULTIPLE_REASONS.

3. If a message is put with one or more MQ header structures at the beginning of the application message data, the queue manager performs certain checks on the header structures to verify that they are valid. For more information about this, see usage note 3 on page 320 for the MQPUT call.
4. If more than one of the warning situations arise (see the *CompCode* parameter), the reason code returned is the *first* one in the following list that applies:
   a. MQRC_MULTIPLE_REASONS
   b. MQRC_INCOMPLETE_MSG
   c. MQRC_INCOMPLETE_GROUP
   d. MQRC_PRIORITY_EXCEEDS_MAXIMUM or MQRC_UNKNOWN_REPORT_OPTION
5. On Tandem NSK, if the MQPUT1 call is issued outside a Tandem TMF transaction *without* the MQPMO_NO_SYNCPOINT option, the reason code MQRC_UNIT_OF_WORK_NOT_STARTED is returned.

C language invocation

```
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 of the message in Buffer */
MQBYTE   Buffer[n];     /* Message data */
MQLONG   CompCode;      /* Completion code */
MQLONG   Reason;        /* Reason code qualifying CompCode */
```

COBOL language invocation

```
     CALL 'MQPUT1' USING HCONN, OBJDESC, MSGDESC, PUTMSGOPTS,
                         BUFFERLENGTH, BUFFER, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN          PIC S9(9) BINARY.
**   Object descriptor
 01  OBJDESC.
     COPY CMQODV.
**   Message descriptor
 01  MSGDESC.
     COPY CMQMDV.
**   Options that control the action of MQPUT1
 01  PUTMSGOPTS.
     COPY CMQPMOV.
**   Length of the message in Buffer
 01  BUFFERLENGTH  PIC S9(9) BINARY.
**   Message data
 01  BUFFER        PIC X(n).
**   Completion code
 01  COMPCODE      PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON        PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength, Buffer,
             CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn         fixed bin(31);  /* Connection handle */
dcl ObjDesc       like MQOD;      /* Object descriptor */
dcl MsgDesc       like MQMD;      /* Message descriptor */
dcl PutMsgOpts    like MQPMO;     /* Options that control the action of
                                     MQPUT1 */
dcl BufferLength  fixed bin(31);  /* Length of the message in Buffer */
dcl Buffer        char(n);        /* Message data */
dcl CompCode      fixed bin(31);  /* Completion code */
dcl Reason        fixed bin(31);  /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
      CALL MQPUT1,(HCONN,OBJDESC,MSGDESC,PUTMSGOPTS,BUFFERLENGTH,       X
            BUFFER,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN             DS      F           Connection handle
OBJDESC           CMQODA              Object descriptor
MSGDESC           CMQMDA              Message descriptor
PUTMSGOPTS        CMQPMOA             Options that control the action
*                                     of MQPUT1
BUFFERLENGTH      DS      F           Length of the message in Buffer
BUFFER            DS      CL(n)       Message data
COMPCODE          DS      F           Completion code
REASON            DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn ;
STRUCT    .EXT ObjDesc(MQOD^Def);
STRUCT    .EXT MsgDesc(MQMD^Def);
STRUCT    .EXT PutMsgOpt(MQPMO^Def);
INT(32)   .EXT BufferLen
STRING    .EXT Buffer[0:BUFFER^SIZE]
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL MQPUT1(HConn, ObjDesc, MsgDesc, PutMsgOpt, BufferLen, Buffer,
           CC, Reason);
```

MQSET – Set object attributes

The MQSET call is used to change the attributes of an object represented by a handle. The object must be a queue.

```
MQSET (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
       CharAttrLength, CharAttrs, CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. The value of *Hconn* was returned by a previous MQCONN or MQCONNX call.

On OS/400, and on MVS/ESA for CICS applications, the MQCONN call can be omitted, and the following value specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

Hobj (MQHOBJ) – input
Object handle.

This handle represents the queue object whose attributes are to be set. The handle was returned by a previous MQOPEN call that specified the MQOO_SET 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 set. Zero is a valid value. The maximum number allowed is 256.

Selectors (MQLONG×*SelectorCount*) – input
Array of attribute selectors.

This is an array of *SelectorCount* attribute selectors; each selector identifies an attribute (integer or character) whose value is to be set.

Each selector must be valid for the type of queue that *Hobj* represents. Only certain MQIA_★ and MQCA_★ values are allowed; these values are listed below.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (MQIA_★ selectors) must be specified in *IntAttrs* in the same order in which these selectors occur in *Selectors*. Attribute values that correspond to character attribute selectors (MQCA_★ selectors) must be specified 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.

It is not an error to specify the same selector more than once; if this is done, the last value specified for a given selector is the one that takes effect.

Notes:

1. The integer and character attribute selectors are allocated within two different ranges; the MQIA_★ selectors reside within the range MQIA_FIRST through MQIA_LAST, and the MQCA_★ selectors within the range MQCA_FIRST through MQCA_LAST.

   For each range, the constants MQIA_LAST_USED and MQCA_LAST_USED define the highest value that the queue manager will accept.

2. If all the MQIA_★ selectors occur first, the same element numbers can be used to address corresponding elements in the *Selectors* and *IntAttrs* arrays.

3. If the *SelectorCount* parameter is zero, *Selectors* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler may be null.

For the MQCA_★ selectors in the following descriptions, the constant that defines the length in bytes of the string that is required in *CharAttrs* is given in parentheses.

Selectors for all types of queue

MQIA_INHIBIT_PUT
: Whether put operations are allowed.

Selectors for local queues

MQCA_TRIGGER_DATA
: Trigger data (MQ_TRIGGER_DATA_LENGTH).

MQIA_DIST_LISTS
: Distribution list support.

MQIA_INHIBIT_GET
: Whether get operations are allowed.

MQIA_TRIGGER_CONTROL
: Trigger control.

MQIA_TRIGGER_DEPTH
: Trigger depth.

MQIA_TRIGGER_MSG_PRIORITY
: Threshold message priority for triggers.

MQIA_TRIGGER_TYPE
: Trigger type.

The selector listed below is supported only in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

MQIA_DIST_LISTS

Selectors for alias queues

MQIA_INHIBIT_GET
: Whether get operations are allowed.

No other attributes can be set using this call.

IntAttrCount (MQLONG) – input
: Count of integer attributes.

This is the number of elements in the *IntAttrs* array, and must be at least

the number of MQIA_* selectors in the *Selectors* parameter. Zero is a valid value if there are none.

IntAttrs (MQLONG×*IntAttrCount*) – input
Array of integer attributes.

This is an array of *IntAttrCount* integer attribute values. These attribute values must be in the same order as the MQIA_* selectors in the *Selectors* array.

If the *IntAttrCount* or *SelectorCount* parameter is zero, *IntAttrs* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler may be null.

CharAttrLength (MQLONG) – input
Length of character attributes buffer.

This is the length in bytes of the *CharAttrs* parameter, and must be at least the sum of the lengths of the character attributes specified in the *Selectors* array. Zero is a valid value if there are no MQCA_* selectors in *Selectors*.

CharAttrs (MQCHAR×*CharAttrLength*) – input
Character attributes.

This is the buffer containing the character attribute values, concatenated together. The length of the buffer is given by the *CharAttrLength* parameter.

The characters attributes must be specified in the same order as the MQCA_* selectors in the *Selectors* array. The length of each character attribute is fixed (see *Selectors*). If the value to be set for an attribute contains fewer nonblank characters than the defined length of the attribute, the value in *CharAttrs* must be padded to the right with blanks to make the attribute value match the defined length of the attribute.

If the *CharAttrLength* or *SelectorCount* parameter is zero, *CharAttrs* is not referred to; in this case, the parameter address passed by programs written in C or System/390 assembler may be null.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_FAILED:

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.

MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.
MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.
MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.
MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.
MQRC_CHAR_ATTR_LENGTH_ERROR
(2006, X'7D6') Length of character attributes not valid.
MQRC_CHAR_ATTRS_ERROR
(2007, X'7D7') Character attributes string not valid.
MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.
MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.
MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.
MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.
MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.
MQRC_INHIBIT_VALUE_ERROR
(2020, X'7E4') Value for inhibit-get or inhibit-put queue attribute not valid.
MQRC_INT_ATTR_COUNT_ERROR
(2021, X'7E5') Count of integer attributes not valid.
MQRC_INT_ATTRS_ARRAY_ERROR
(2023, X'7E7') Integer attributes array not valid.
MQRC_NOT_OPEN_FOR_SET
(2040, X'7F8') Queue not open for set.
MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.
MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.
MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.
MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.
MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.
MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.
MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SELECTOR_COUNT_ERROR
(2065, X'811') Count of selectors not valid.
MQRC_SELECTOR_ERROR
(2067, X'813') Attribute selector not valid.

MQRC_SELECTOR_LIMIT_EXCEEDED
(2066, X'812') Count of selectors too big.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.
MQRC_TRIGGER_CONTROL_ERROR
(2075, X'81B') Value for trigger-control attribute not valid.
MQRC_TRIGGER_DEPTH_ERROR
(2076, X'81C') Value for trigger-depth attribute not valid.
MQRC_TRIGGER_MSG_PRIORITY_ERR
(2077, X'81D') Value for trigger-message-priority attribute not valid.
MQRC_TRIGGER_TYPE_ERROR
(2078, X'81E') Value for trigger-type attribute not valid.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

Usage notes

1. Using this call, the application can specify an array of integer attributes, or a collection of character attribute strings, or both. The attributes specified are all set simultaneously, if no errors occur. If an error does occur (for example, if a selector is not valid, or an attempt is made to set an attribute to a value that is not valid), the call fails and no attributes are set.
2. The values of attributes can be determined using the MQINQ call; see “MQINQ – Inquire about object attributes” on page 285 for details.

   Note: Not all attributes whose values can be inquired using the MQINQ call can have their values changed using the MQSET call. For example, no process-object or queue-manager attributes can be set with this call.
3. Attribute changes are preserved across restarts of the queue manager (other than alterations to temporary dynamic queues, which do not survive restarts of the queue manager).
4. It is not possible to change the attributes of a model queue using the MQSET call. However, if you open a model queue using the MQOPEN call with the MQOO_SET option, you can use the MQSET call to set the attributes of the dynamic queue that is created by the MQOPEN call.
5. For more information about object attributes, see Chapter 4, “Attributes of MQSeries objects.”

C language invocation

```
MQSET (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 */
```

COBOL language invocation

```
     CALL 'MQSET' USING HCONN, HOBJ, SELECTORCOUNT,
                        SELECTORS-TABLE, INTATTRCOUNT, INTATTRS-TABLE,
                        CHARATTRLENGTH, CHARATTRS, COMPCODE, REASON.
```

Declare the parameters as follows:

```
**   Connection handle
 01  HCONN            PIC S9(9) BINARY.
**   Object handle
 01  HOBJ             PIC S9(9) BINARY.
**   Count of selectors
 01  SELECTORCOUNT    PIC S9(9) BINARY.
**   Array of attribute selectors
 01  SELECTORS-TABLE.
  02  SELECTORS        PIC S9(9) BINARY OCCURS n TIMES.
**   Count of integer attributes
 01  INTATTRCOUNT     PIC S9(9) BINARY.
**   Array of integer attributes
 01  INTATTRS-TABLE.
  02  INTATTRS         PIC S9(9) BINARY OCCURS n TIMES.
**   Length of character attributes buffer
 01  CHARATTRLENGTH   PIC S9(9) BINARY.
**   Character attributes
 01  CHARATTRS        PIC X(n).
**   Completion code
 01  COMPCODE         PIC S9(9) BINARY.
**   Reason code qualifying CompCode
 01  REASON           PIC S9(9) BINARY.
```

PL/I language invocation (AIX, MVS/ESA, OS/2, and Windows NT)

```
call MQSET (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount,
            IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason);
```

Declare the parameters as follows:

```
dcl Hconn           fixed bin(31);  /* Connection handle */
dcl Hobj            fixed bin(31);  /* Object handle */
dcl SelectorCount   fixed bin(31);  /* Count of selectors */
dcl Selectors(n)    fixed bin(31);  /* Array of attribute selectors */
dcl IntAttrCount    fixed bin(31);  /* Count of integer attributes */
dcl IntAttrs(n)     fixed bin(31);  /* Array of integer attributes */
dcl CharAttrLength  fixed bin(31);  /* Length of character attributes
                                       buffer */
dcl CharAttrs       char(n);        /* Character attributes */
dcl CompCode        fixed bin(31);  /* Completion code */
dcl Reason          fixed bin(31);  /* Reason code qualifying
                                       CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
      CALL MQSET,(HCONN,HOBJ,SELECTORCOUNT,SELECTORS,INTATTRCOUNT,      X
            INTATTRS,CHARATTRLENGTH,CHARATTRS,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
HOBJ                DS      F           Object handle
SELECTORCOUNT       DS      F           Count of selectors
SELECTORS           DS      (n)F        Array of attribute selectors
INTATTRCOUNT        DS      F           Count of integer attributes
INTATTRS            DS      (n)F        Array of integer attributes
CHARATTRLENGTH      DS      F           Length of character attributes
*                                       buffer
CHARATTRS           DS      CL(n)       Character attributes
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

TAL invocation (Tandem NSK only)

```
INT(32)   .EXT HConn ;
INT(32)   .EXT HObj;
INT(32)      SelectorCount;
INT(32)   .EXT Selectors[0:NUM^SELECTORS];
INT(32)      IntAttrCount;
INT(32)   .EXT IntAttrs[0:NUM^INT^ATTR];
INT(32)      CharAttrLength;
STRING    .EXT CharAttrs[0:LEN^CHAR^ATTR];
INT(32)   .EXT CC;
INT(32)   .EXT Reason;

CALL MQSET(HConn, HObj, SelectorCount, Selectors, IntAttrCount, IntAttrs,
                          CharAttrLength, CharAttrs, CC, Reason);
```

MQSYNC – Synchronize statistics updates (Tandem NSK only)

The MQSYNC call is included in this release of MQSeries for Tandem NonStop Kernel for backwards compatibility with MQSeries for Tandem NSK, Version 1.5.1. but performs no function.

The call always returns a *CompCode* of MQCC_OK, and a *Reason* of MQRC_NONE.

```
MQSYNC (TransId, CommitAbort, CompCode, Reason)
```

Parameters

TransId (MQCHAR48) – input
Transaction identifier.

CommitAbort (MQCHAR48) – input
Commit flag.

CompCode (MQLONG) – output
Completion code.

It is the following:

MQCC_OK
Successful completion.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

For *CompCode* of MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

C language invocation

```
transaction_id_def TransID;
int         CommitAbort;
MQLONG      CompCode;
MQLONG      Reason;

MQSYNC(&TransID, CommitAbort, &CompCode, &Reason);
```

MQSYNC – COBOL invocation

```
01 TRANSID       NATIVE-4.
01 COMMITABORT   NATIVE-4.
01 COMPCODE      NATIVE-4.
01 REASON        NATIVE-4.

CALL 'MQSYNC' USING TRANSID COMMITABORT.
```

MQSYNC – TAL invocation

```
STRING   .EXT TransID;
INT CommitAbort;
INT(32)  .EXT CC;
INT(32)  .EXT Reason;

CALL MQSYNC(TransID, CommitAbort, CC, Reason);
```

Chapter 4. Attributes of MQSeries objects

MQSeries objects consist of:

- Channels
- Queues
- Queue managers
- Namelists (MVS/ESA only)
- Processes
- Storage Classes (MVS/ESA only)

This chapter describes the attributes (or properties) of MQSeries objects that are accessible through the API, which are queues, queue managers, namelists, and processes. The attributes are grouped according to the type of object to which they apply; see:

- "Attributes for all queues"
- "Attributes for local queues and model queues" on page 348
- "Attributes for local definitions of remote queues" on page 363
- "Attributes for alias queues" on page 365
- "Attributes for namelists (MVS/ESA only)" on page 366
- "Attributes for process definitions" on page 367
- "Attributes for the queue manager" on page 370

Within each section, the attributes are listed in alphabetic order.

Note: The names of the attributes of objects are shown in this book in the form that you use them with the MQINQ and MQSET calls. When you use MQSeries commands to define, alter, or display the attributes, you use the keywords shown in the descriptions of the commands in the *MQSeries Command Reference*.

Attributes for all queues

The following table summarizes the attributes that are common to all queue types (except where noted). The attributes are described in alphabetic order.

Table 61. Attributes for all queues

Attribute	Description	Page
DefPersistence	Default message persistence	344
DefPriority	Default message priority	344
InhibitGet	Controls whether get operations for the queue are allowed	345
InhibitPut	Controls whether put operations for the queue are allowed	345
QDesc	Queue description	346
QName	Queue name	346
QType	Queue type	346
Scope	Controls whether an entry for the queue also exists in a cell directory	346

DefPersistence (MQLONG)
Default message persistence.

This is the default persistence for messages on a queue. This applies if MQPER_PERSISTENCE_AS_Q_DEF is specified in the message descriptor when the message is put.

If there is more than one definition in the queue-name resolution path, the default persistence is taken from the value of this attribute in the *first* definition in the path at the time of the put operation (even if this is a queue-manager alias).

The value is one of the following:

MQPER_PERSISTENT
Message is persistent.

The message survives restarts of the queue manager. Because temporary dynamic queues *do not* survive restarts of the queue manager, persistent messages cannot be put on temporary dynamic queues; persistent messages can however be put on permanent dynamic queues, and predefined queues.

MQPER_NOT_PERSISTENT
Message is not persistent.

The message does not survive restarts of the queue manager. This applies even if an intact copy of the message is found on auxiliary storage during the restart procedure.

Both persistent and nonpersistent messages can exist on the same queue.

To determine the value of this attribute, use the MQIA_DEF_PERSISTENCE selector with the MQINQ call.

DefPriority (MQLONG)
Default message priority

This is the default priority for messages on the queue. This applies if MQPRI_PRIORITY_AS_Q_DEF is specified in the message descriptor when the message is put on the queue.

If there is more than one definition in the queue-name resolution path, the default priority for the message is taken from the value of this attribute in the *first* definition in the path at the time of the put operation (even if this is a queue-manager alias).

The way in which a message is placed on a queue depends on the value of the queue's *MsgDeliverySequence* attribute:

- If the *MsgDeliverySequence* attribute is MQMDS_PRIORITY, the logical position at which a message is placed on the queue is dependent on the value of the *Priority* field in the message descriptor.
- If the *MsgDeliverySequence* attribute is MQMDS_FIFO, messages are placed on the queue as though they had a priority equal to the *DefPriority* of the resolved queue, regardless of the value of the *Priority* field in the message descriptor. However, the *Priority* field retains the value specified by the application that put the message. See the *MsgDeliverySequence* attribute described in "Attributes for local queues and model queues" on page 348 for more information.

Priorities are in the range zero (lowest) through *MaxPriority* (highest); see the *MaxPriority* attribute described in “Attributes for the queue manager” on page 370.

To determine the value of this attribute, use the MQIA_DEF_PRIORITY selector with the MQINQ call.

InhibitGet (MQLONG)
Controls whether get operations for this queue are allowed.

This attribute applies only to local, model, and alias queues.

If the queue is an alias queue, get operations must be allowed for both the alias and the base queue at the time of the get operation, in order for the MQGET call to succeed.

The value is one of the following:

MQQA_GET_INHIBITED
Get operations are inhibited.

MQGET calls fail with reason code MQRC_GET_INHIBITED. This includes MQGET calls that specify MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT.

Note: If an MQGET call operating within a unit of work completes successfully, changing the value of the *InhibitGet* attribute subsequently to MQQA_GET_INHIBITED does not prevent the unit of work being committed.

MQQA_GET_ALLOWED
Get operations are allowed.

To determine the value of this attribute, use the MQIA_INHIBIT_GET selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

InhibitPut (MQLONG)
Controls whether put operations for this queue are allowed.

If there is more than one definition in the queue-name resolution path, put operations must be allowed for *every* definition in the path (including any queue-manager alias definitions) at the time of the put operation, in order for the MQPUT or MQPUT1 call to succeed.

The value is one of the following:

MQQA_PUT_INHIBITED
Put operations are inhibited.

MQPUT and MQPUT1 calls fail with reason code MQRC_PUT_INHIBITED.

Note: If an MQPUT call operating within a unit of work completes successfully, changing the value of the *InhibitPut* attribute subsequently to MQQA_PUT_INHIBITED does not prevent the unit of work being committed.

MQQA_PUT_ALLOWED
Put operations are allowed.

To determine the value of this attribute, use the MQIA_INHIBIT_PUT selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

QDesc (MQCHAR64)
Queue description.

This is a field that may be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager may require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters may be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the MQCA_Q_DESC selector with the MQINQ call. The length of this attribute is given by MQ_Q_DESC_LENGTH.

QName (MQCHAR48)
Queue name.

This is the name of a queue defined on the local queue manager. For more information about queue names, see the *MQSeries Application Programming Guide*. All queues defined on a queue manager share the same queue name space. Therefore, a MQQT_LOCAL queue and a MQQT_ALIAS queue cannot have the same name.

To determine the value of this attribute, use the MQCA_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

QType (MQLONG)
Queue type.

This attribute has one of the following values:

MQQT_ALIAS
Alias queue definition.
MQQT_LOCAL
Local queue.
MQQT_MODEL
Model queue definition.
MQQT_REMOTE
Local definition of a remote queue.

To determine the value of this attribute, use the MQIA_Q_TYPE selector with the MQINQ call.

Scope (MQLONG)
Controls whether an entry for this queue also exists in a cell directory.

A cell directory is provided by an installable Name service. This attribute applies only to local and alias queues, and to local definitions of remote queues. It does not apply to model queues.

The value is one of the following:

MQSCO_Q_MGR
: Queue-manager scope.

  The queue definition has queue-manager scope. This means that the definition of the queue does not extend beyond the queue manager which owns it. To open the queue for output from some other queue manager, either the name of the owning queue manager must be specified, or the other queue manager must have a local definition of the queue.

MQSCO_CELL
: Cell scope.

  The queue definition has cell scope. This means that the queue definition is also placed in a cell directory available to all of the queue managers in the cell. The queue can be opened for output from any of the queue managers in the cell merely by specifying the name of the queue; the name of the queue manager which owns the queue need not be specified. However, the cell definition is not available to any queue manager in the cell which also has a local definition of a queue with that name, as the local definition takes precedence.

  A cell directory is provided by an installable Name service. For example, the DCE Name service inserts the queue definition into the DCE directory.

  Model and dynamic queues cannot have cell scope.

  This value is only valid if a name service supporting a cell directory has been configured.

To determine the value of this attribute, use the MQIA_SCOPE selector with the MQINQ call.

Support for this attribute is subject to the following restrictions:

- On OS/400, the attribute is supported, but only MQSCO_Q_MGR is valid.
- On MVS/ESA, 16-bit Windows, and 32-bit Windows, the attribute is not supported.

Attributes for local queues and model queues

The following table summarizes the attributes that are specific to local queues and model queues (except where noted). The attributes are described in alphabetic order.

Table 62 (Page 1 of 2). Attributes for local and model queues

Attribute	Description	Page
BackoutRequeueQName	Excessive backout requeue queue name	349
BackoutThreshold	Backout threshold	349
CreationDate	Date the queue was created	349
CreationTime	Time the queue was created	349
CurrentQDepth	Current queue depth	350
DefinitionType	Queue definition type	350
DefInputOpenOption	Default input open option	351
DistLists	Distribution list support	351
HardenGetBackout	Whether to maintain an accurate backout count	352
IndexType	Index type	353
InitiationQName	Name of initiation queue	354
MaxMsgLength	Maximum message length in bytes	354
MaxQDepth	Maximum queue depth	355
MsgDeliverySequence	Message delivery sequence	355
OpenInputCount	Number of opens for input	356
OpenOutputCount	Number of opens for output	356
ProcessName	Process name	357
QDepthHighEvent	Controls whether Queue Depth High events are generated	357
QDepthHighLimit	High limit for queue depth	357
QDepthLowEvent	Controls whether Queue Depth Low events are generated	358
QDepthLowLimit	Low limit for queue depth	358
QDepthMaxEvent	Controls whether Queue Full events are generated	358
QServiceInterval	Target for queue service interval	359
QServiceIntervalEvent	Controls whether Service Interval High or Service Interval OK events are generated	359
RetentionInterval	Retention interval	360
Shareability	Queue shareability	360
StorageClass	Storage class for queue	360
TriggerControl	Trigger control	361
TriggerData	Trigger data	361
TriggerDepth	Trigger depth	361
TriggerMsgPriority	Threshold message priority for triggers	362

Table 62 (Page 2 of 2). Attributes for local and model queues		
Attribute	**Description**	**Page**
TriggerType	Trigger type	362
Usage	Queue usage	363

BackoutRequeueQName (MQCHAR48)
Excessive backout requeue queue name.

Apart from allowing its value to be queried, the queue manager takes no action based on the value of this attribute.

To determine the value of this attribute, use the MQCA_BACKOUT_REQ_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

BackoutThreshold (MQLONG)
Backout threshold.

Apart from allowing its value to be queried, the queue manager takes no action based on the value of this attribute.

To determine the value of this attribute, use the MQIA_BACKOUT_THRESHOLD selector with the MQINQ call.

CreationDate (MQCHAR12)
Date this queue was created.

The format is

```
YYYY-MM-DD
```

with 2 bytes of blank padding to the right to make the length 12 bytes. For example:

```
1992-09-23bb
```

is 23 September 1992 ("bb" represents 2 blank characters).

On OS/400, the creation date of a queue may differ from that of the underlying operating system entity (file or userspace) that represents the queue.

To determine the value of this attribute, use the MQCA_CREATION_DATE selector with the MQINQ call. The length of this attribute is given by MQ_CREATION_DATE_LENGTH.

CreationTime (MQCHAR8)
Time this queue was created.

The format is

```
HH.MM.SS
```

using the 24-hour clock, with a leading zero if the hour is less than 10. For example:

```
21.10.20
```

This is an 8-character string. The time is local time.

- On MVS/ESA, the time is Greenwich Mean Time (GMT), subject to the system clock being set accurately to GMT.

- On OS/400, the creation time of a queue may differ from that of the underlying operating system entity (file or userspace) that represents the queue.

To determine the value of this attribute, use the MQCA_CREATION_TIME selector with the MQINQ call. The length of this attribute is given by MQ_CREATION_TIME_LENGTH.

CurrentQDepth (MQLONG)
Current queue depth.

This is the number of messages currently on the queue. It is incremented during an MQPUT call, and during backout of an MQGET call. It is decremented during a nonbrowse MQGET call, and during backout of an MQPUT call. The effect of this is that the count includes messages that have been put on the queue within a unit of work, but which have not yet been committed, even though they are not eligible to be retrieved by the MQGET call. Similarly, it excludes messages that have been retrieved within a unit of work using the MQGET call, but which have yet to be committed.

The count also includes messages which have passed their expiry time but have not yet been discarded, although these messages are not eligible to be retrieved. See the *Expiry* field described in “MQMD – Message descriptor” on page 98.

The value of this attribute fluctuates as the queue manager operates.

This attribute does not apply to model queues, but it does apply to the dynamically-defined queues created from the model queue definitions using the MQOPEN call.

To determine the value of this attribute, use the MQIA_CURRENT_Q_DEPTH selector with the MQINQ call.

DefinitionType (MQLONG)
Queue definition type.

This indicates how the queue was defined. It is one of the following:

MQQDT_PREDEFINED
Predefined permanent queue.

The queue is a permanent queue created by the system administrator; only the system administrator can delete it.

Predefined queues are created using the DEFINE command, and can be deleted only by using the DELETE command. Predefined queues cannot be created from model queues.

Commands can be issued either by an operator, or by an authorized application sending a command message to the command input queue (see the *CommandInputQName* attribute described in “Attributes for the queue manager” on page 370).

MQQDT_PERMANENT_DYNAMIC
Dynamically defined permanent queue.

The queue is a permanent queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor. The model queue definition has the value

MQQDT_PERMANENT_DYNAMIC for the *DefinitionType* attribute. This type of queue can be deleted using the MQCLOSE call. See "MQCLOSE – Close object" on page 248 for more details.

MQQDT_TEMPORARY_DYNAMIC
Dynamically defined temporary queue.

The queue is a temporary queue that was created by an application issuing an MQOPEN call with the name of a model queue specified in the object descriptor. The model queue definition has the value MQQDT_TEMPORARY_DYNAMIC for the *DefinitionType* attribute. This type of queue is deleted automatically by the MQCLOSE call when it is closed by the application that created it.

This attribute in a model queue definition does not indicate how the model queue was defined, because model queues are always predefined. Instead, the value of this attribute in the model queue is used to determine the *DefinitionType* of each of the dynamic queues created from the model queue definition using the MQOPEN call.

To determine the value of this attribute, use the MQIA_DEFINITION_TYPE selector with the MQINQ call.

DefInputOpenOption (MQLONG)
Default input open option.

This is the default way in which the queue should be opened for input. It applies if the MQOO_INPUT_AS_Q_DEF option is specified on the MQOPEN call when the queue is opened. It is one of the following:

MQOO_INPUT_EXCLUSIVE
Open queue to get messages with exclusive access.

The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open by this or another application for input of any type (MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE).

MQOO_INPUT_SHARED
Open queue to get messages with shared access.

The queue is opened for use with subsequent MQGET calls. The call can succeed if the queue is currently open by this or another application with MQOO_INPUT_SHARED, but fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open with MQOO_INPUT_EXCLUSIVE.

To determine the value of this attribute, use the MQIA_DEF_INPUT_OPEN_OPTION selector with the MQINQ call.

DistLists (MQLONG)
Distribution list support.

This indicates whether distribution-list messages can be placed on the queue. The attribute is set by a message channel agent (MCA) to inform the local queue manager whether the queue manager at the other end of the channel supports distribution lists. This latter queue manager (called the "partnering queue manager") is the one which next receives the message, after it has been removed from the local transmission queue by a sending MCA.

The attribute is set by the sending MCA whenever it establishes a connection to the receiving MCA on the partnering queue manager. In this way, the sending MCA can cause the local queue manager to place on the transmission queue only messages which the partnering queue manager is capable of processing correctly.

This attribute is primarily for use with transmission queues, but the processing described is performed regardless of the usage defined for the queue (see the *Usage* attribute).

The value is one of the following:

MQDL_SUPPORTED
: Distribution lists supported.

  This indicates that distribution-list messages can be stored on the queue, and transmitted to the partnering queue manager in that form. This reduces the amount of processing required to send the message to multiple destinations.

MQDL_NOT_SUPPORTED
: Distribution lists not supported.

  This indicates that distribution-list messages cannot be stored on the queue, because the partnering queue manager does not support distribution lists. If an application puts a distribution-list message, and that message is to be placed on this queue, the queue manager splits the distribution-list message and places the individual messages on the queue instead. This increases the amount of processing required to send the message to multiple destinations, but ensures that the messages will be processed correctly by the partnering queue manager.

To determine the value of this attribute, use the MQIA_DIST_LISTS selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

HardenGetBackout (MQLONG)
: Whether to maintain an accurate backout count.

  For each message, a count is kept of the number of times that the message is retrieved by an MQGET call within a unit of work, and that unit of work subsequently backed out. This count is available in the *BackoutCount* field in the message descriptor after the MQGET call has completed.

  The message backout count survives restarts of the queue manager. However, to ensure that the count is accurate, information has to be "hardened" (recorded on disk or other permanent storage device) each time a message is retrieved by an MQGET call within a unit of work for this queue. If this is not done, and a failure of the queue manager occurs together with backout of the MQGET call, the count may or may not be incremented.

  Hardening information for each MQGET call within a unit of work, however, imposes a performance overhead, and the *HardenGetBackout*

attribute should be set to MQQA_BACKOUT_HARDENED only if it is essential that the count is accurate.

On OpenVMS, OS/2, OS/400, Tandem NSK, UNIX systems, and Windows NT, the message backout count is always hardened, regardless of the setting of this attribute.

The following values are possible:

MQQA_BACKOUT_HARDENED
: Backout count remembered.

  Hardening is used to ensure that the backout count for messages on this queue is accurate.

MQQA_BACKOUT_NOT_HARDENED
: Backout count may not be remembered.

  Hardening is not used to ensure that the backout count for messages on this queue is accurate. The count may therefore be lower than it should be.

To determine the value of this attribute, use the MQIA_HARDEN_GET_BACKOUT selector with the MQINQ call.

IndexType (MQLONG)
: Index type.

  This specifies the type of index that the queue manager maintains in order to speed MQGET operations on the queue. No single value is optimal for all queues – it depends on how the messages on the queue are retrieved by the application.

  The value is one of the following:

  MQIT_NONE
  : No index.

    No index is maintained by the queue manager for this queue. This is the value that should be used for queues which are usually processed sequentially, that is, without using any selection criteria on the MQGET call.

  MQIT_MSG_ID
  : Queue is indexed using message identifiers.

    The queue manager maintains an index that uses the message identifiers of the messages on the queue. This is the value that should be used for queues where the application usually retrieves messages using the message identifier as the selection criterion on the MQGET call (that is, the application usually specifies a value *other than* MQMI_NONE for the *MsgId* field in the MQMD structure).

  MQIT_CORREL_ID
  : Queue is indexed using correlation identifiers.

    The queue manager maintains an index that uses the correlation identifiers of the messages on the queue. This is the value that should be used for queues where the application usually retrieves messages using the correlation identifier as the selection criterion on the MQGET call (that is, the application usually specifies a value

other than MQCI_NONE for the *CorrelId* field in the MQMD structure).

Applications can retrieve messages from the queue regardless of the value of this attribute; its purpose is merely to improve performance in those situations where the application processes the queue in one of the ways described above.

To determine the value of this attribute, use the MQIA_INDEX_TYPE selector with the MQINQ call.

This attribute is supported only on MVS/ESA. On other platforms retrieval optimization may be provided, but it is not controlled by an attribute.

InitiationQName (MQCHAR48)
Name of initiation queue.

This is the name of a queue defined on the local queue manager; the queue must be of type MQQT_LOCAL. The queue manager sends a trigger message to the initiation queue when application start-up is required as a result of a message arriving on the queue to which this attribute belongs. The initiation queue must be monitored by a trigger monitor application which will start the appropriate application after receipt of the trigger message.

To determine the value of this attribute, use the MQCA_INITIATION_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

MaxMsgLength (MQLONG)
Maximum message length in bytes.

This is the maximum length of the application message data that can exist in each message on the queue. The *MaxMsgLength* local-queue attribute can be set independently of the *MaxMsgLength* queue-manager attribute, and the longest physical message that can be placed on a queue is the lesser of those two values. An attempt to place on the queue a message that is too long fails with reason code:

- MQRC_MSG_TOO_BIG_FOR_Q if the message to too big for the queue
- MQRC_MSG_TOO_BIG_FOR_Q_MGR if the message to too big for the queue manager, but not too big for the queue

The value of this attribute is greater than or equal to zero. The upper limit is determined by the environment:

- On AIX, HP-UX, OS/2, Sun Solaris, and Windows NT, the maximum message length is 100 MB (104 857 600 bytes).
- On OpenVMS, MVS/ESA, OS/400, Tandem NSK, UNIX systems not listed above, 16-bit Windows, and 32-bit Windows, the maximum message length is 4 MB (4 194 304 bytes).

For more information, see the *BufferLength* parameter described in “MQPUT – Put message” on page 313.

To determine the value of this attribute, use the MQIA_MAX_MSG_LENGTH selector with the MQINQ call.

MaxQDepth (MQLONG)
Maximum queue depth.

This is the defined upper limit for the number of physical messages that can exist on the queue at any one time. An attempt to put a message on a queue that already contains *MaxQDepth* messages fails with reason code MQRC_Q_FULL.

Note: Unit-of-work processing and the segmentation of messages can both cause the actual number of physical messages on the queue to exceed *MaxQDepth*. However, this does not affect the retrievability of the messages – *all* messages on the queue can be retrieved using the MQGET call in the normal way.

The value of this attribute is zero or greater. The upper limit is determined by the environment:

- On OpenVMS, OS/2, OS/400, Tandem NSK, UNIX systems, and Windows NT, the value cannot exceed 640 000.

Note: It is possible for the storage space available to the queue to be exhausted even if there are fewer than *MaxQDepth* messages on the queue.

To determine the value of this attribute, use the MQIA_MAX_Q_DEPTH selector with the MQINQ call.

MsgDeliverySequence (MQLONG)
Message delivery sequence.

This determines the order in which messages are returned to the application by the MQGET call:

MQMDS_PRIORITY
Messages are returned in priority order.

This means that an MQGET call will return the *highest-priority* message that satisfies the selection criteria specified on the call. Within each priority level, messages are returned in FIFO order (first in, first out).

MQMDS_FIFO
Messages are returned in FIFO order (first in, first out).

This means that an MQGET call will return the *first* message that satisfies the selection criteria specified on the call, regardless of priority.

If the relevant attributes are changed while there are messages on the queue, the delivery sequence is as follows:

The order in which messages are returned by the MQGET call is determined by the values of the *MsgDeliverySequence* and *DefPriority* attributes in force for the queue at the time the message arrives on the queue:

- If *MsgDeliverySequence* is MQMDS_FIFO when the message arrives, the message is placed on the queue as though its priority were *DefPriority*.
- If *MsgDeliverySequence* is MQMDS_PRIORITY when the message arrives, the message is placed on the queue at the place appropriate to priority given by the *Priority* field in the message descriptor.

If the value of the *MsgDeliverySequence* attribute is subsequently changed while there are messages on the queue, the order of the messages on the queue is not changed. This does not affect the value of the *Priority* field in the MQMD, which retains the value it had when the message was first put.

This means that if the value of the *DefPriority* attribute is changed, messages will not necessarily be delivered in FIFO order, even though the *MsgDeliverySequence* attribute is set to MQMDS_FIFO; those that were placed on the queue at the higher priority are delivered first.

To determine the value of this attribute, use the MQIA_MSG_DELIVERY_SEQUENCE selector with the MQINQ call.

OpenInputCount (MQLONG)
Number of opens for input.

This is the number of handles that are currently valid for removing messages from the queue by means of the MQGET call. It is the total number of such handles known to the local queue manager.

The count includes handles where an alias queue which resolves to this queue was opened for input. The count does not include handles where the queue was opened for action(s) which did not include input (for example, a queue opened only for browse).

The value of this attribute fluctuates as the queue manager operates.

This attribute does not apply to model queues, but it does apply to the dynamically-defined queues created from the model queue definitions using the MQOPEN call.

To determine the value of this attribute, use the MQIA_OPEN_INPUT_COUNT selector with the MQINQ call.

OpenOutputCount (MQLONG)
Number of opens for output.

This is the number of handles that are currently valid for adding messages to the queue by means of the MQPUT call. It is the total number of such handles known to the *local* queue manager; it does not include opens for output that were performed for this queue at remote queue managers.

The count includes handles where an alias queue which resolves to this queue was opened for output. The count does not include handles where the queue was opened for action(s) which did not include output (for example, a queue opened only for inquire).

The value of this attribute fluctuates as the queue manager operates.

This attribute does not apply to model queues, but it does apply to the dynamically-defined queues created from the model queue definitions using the MQOPEN call.

To determine the value of this attribute, use the MQIA_OPEN_OUTPUT_COUNT selector with the MQINQ call.

ProcessName (MQCHAR48)
Process name.

This is the name of a process object that is defined on the local queue manager. The process object identifies a program that can service the queue.

To determine the value of this attribute, use the MQCA_PROCESS_NAME selector with the MQINQ call. The length of this attribute is given by MQ_PROCESS_NAME_LENGTH.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

QDepthHighEvent (MQLONG)
Controls whether Queue Depth High events are generated.

A Queue Depth High event indicates that an application has put a message on a queue, and this has caused the number of messages on the queue to become greater than or equal to the queue depth high threshold (see the *QDepthHighLimit* attribute).

Note: The value of this attribute can change dynamically. See the description of the Queue Depth High event for more details.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_Q_DEPTH_HIGH_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

QDepthHighLimit (MQLONG)
High limit for queue depth.

The threshold against which the queue depth is compared to generate a Queue Depth High event.

This event indicates that an application has put a message on a queue, and this has caused the number of messages on the queue to become greater than or equal to the queue depth high threshold. See the *QDepthHighEvent* attribute.

The value is expressed as a percentage of the maximum queue depth (*MaxQDepth* attribute), and is greater than or equal to 0 and less than or equal to 100. The default value is 80.

To determine the value of this attribute, use the MQIA_Q_DEPTH_HIGH_LIMIT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

QDepthLowEvent (MQLONG)
Controls whether Queue Depth Low events are generated.

A Queue Depth Low event indicates that an application has retrieved a message from a queue, and this has caused the number of messages on the queue to become less than or equal to the queue depth low threshold (see the *QDepthLowLimit* attribute).

Note: The value of this attribute can change dynamically. See the description of the Queue Depth Low event for more details.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_Q_DEPTH_LOW_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

QDepthLowLimit (MQLONG)
Low limit for queue depth.

The threshold against which the queue depth is compared to generate a Queue Depth Low event.

This event indicates that an application has retrieved a message from a queue, and this has caused the number of messages on the queue to become less than or equal to the queue depth low threshold. See the *QDepthLowEvent* attribute.

The value is expressed as a percentage of the maximum queue depth (*MaxQDepth* attribute), and is greater than or equal to 0 and less than or equal to 100. The default value is 20.

To determine the value of this attribute, use the MQIA_Q_DEPTH_LOW_LIMIT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

QDepthMaxEvent (MQLONG)
Controls whether Queue Full events are generated.

A Queue Full event indicates that a put to a queue has been rejected because the queue is full, that is, the queue depth has already reached its maximum value.

Note: The value of this attribute can change dynamically. See the description of the Queue Full event for more details.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_Q_DEPTH_MAX_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

QServiceInterval (MQLONG)
Target for queue service interval.

The service interval used for comparison to generate Service Interval High and Service Interval OK events. See the *QServiceIntervalEvent* attribute.

The value is in units of milliseconds, and is greater than or equal to zero, and less than or equal to 999 999 999.

To determine the value of this attribute, use the MQIA_Q_SERVICE_INTERVAL selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

QServiceIntervalEvent (MQLONG)
Controls whether Service Interval High or Service Interval OK events are generated.

A Service Interval High event is generated when a check indicates that no messages have been retrieved from the queue for at least the time indicated by the *QServiceInterval* attribute.

A Service Interval OK event is generated when a check indicates that messages have been retrieved from the queue within the time indicated by the *QServiceInterval* attribute.

Note: The value of this attribute can change dynamically. See the description of the Service Interval High and Service Interval OK events for more details.

It is one of the following:

MQQSIE_HIGH
Queue Service Interval High events enabled.

- Queue Service Interval High events are **enabled** and
- Queue Service Interval OK events are **disabled**.

MQQSIE_OK
Queue Service Interval OK events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are **enabled**.

MQQSIE_NONE
No queue service interval events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are also **disabled**.

To determine the value of this attribute, use the MQIA_Q_SERVICE_INTERVAL_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

RetentionInterval (MQLONG)
Retention interval.

This is the period of time for which the queue should be retained. After this time has elapsed, the queue is eligible for deletion.

The time is measured in hours, counting from the date and time when the queue was created. The creation date and time of the queue are recorded in the *CreationDate* and *CreationTime* attributes, respectively.

This information is provided to enable a housekeeping application or the operator to identify and delete queues that are no longer required.

Note: The queue manager never takes any action to delete queues based on this attribute, or to prevent the deletion of queues whose retention interval has not expired; it is the user's responsibility to cause any required action to be taken.

A realistic retention interval should be used to prevent the accumulation of permanent dynamic queues (see *DefinitionType*). However, this attribute can also be used with predefined queues.

To determine the value of this attribute, use the MQIA_RETENTION_INTERVAL selector with the MQINQ call.

Shareability (MQLONG)
Queue shareability.

This indicates whether the queue can be opened for input multiple times concurrently. It is one of the following:

MQQA_SHAREABLE
Queue is shareable.

Multiple opens with the MQOO_INPUT_SHARED option are allowed.

MQQA_NOT_SHAREABLE
Queue is not shareable.

An MQOPEN call with the MQOO_INPUT_SHARED option is treated as MQOO_INPUT_EXCLUSIVE.

To determine the value of this attribute, use the MQIA_SHAREABILITY selector with the MQINQ call.

StorageClass (MQCHAR8)
Storage class for queue.

This is a user-defined name that defines the physical storage used to hold the queue. In practice, a message is written to disk only if it needs to be paged out of its memory buffer.

To determine the value of this attribute, use the MQCA_STORAGE_CLASS selector with the MQINQ call. The length of this attribute is given by MQ_STORAGE_CLASS_LENGTH.

This attribute is supported only on MVS/ESA.

TriggerControl (MQLONG)
Trigger control.

This controls whether trigger messages are written to an initiation queue, in order to cause an application to be started to service the queue.

This is one of the following:

MQTC_OFF
Trigger messages not required.

No trigger messages are to be written for this queue. The value of *TriggerType* is irrelevant in this case.

MQTC_ON
Trigger messages required.

Trigger messages are to be written for this queue, when the appropriate trigger events occur.

To determine the value of this attribute, use the MQIA_TRIGGER_CONTROL selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

TriggerData (MQCHAR64)
Trigger data.

This is free-format data that the queue manager inserts into the trigger message when a message arriving on this queue causes a trigger message to be written to the initiation queue.

The content of this data is of no significance to the queue manager. It is meaningful either to the trigger-monitor application which processes the initiation queue, or to the application which is started by the trigger monitor.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_TRIGGER_DATA selector with the MQINQ call. To change the value of this attribute, use the MQSET call. The length of this attribute is given by MQ_TRIGGER_DATA_LENGTH.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

TriggerDepth (MQLONG)
Trigger depth.

This is the number of messages that have to be on the queue before a trigger message is written when *TriggerType* is set to MQTT_DEPTH. The value of *TriggerDepth* is one or greater. This attribute is not used otherwise.

To determine the value of this attribute, use the MQIA_TRIGGER_DEPTH selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

TriggerMsgPriority (MQLONG)
Threshold message priority for triggers.

This is the message priority below which messages do not contribute to the generation of trigger messages (that is, the queue manager ignores these messages when deciding whether a trigger message should be generated). *TriggerMsgPriority* can be in the range zero (lowest) through *MaxPriority* (highest; see "Attributes for the queue manager" on page 370); a value of zero causes all messages to contribute to the generation of trigger messages.

To determine the value of this attribute, use the MQIA_TRIGGER_MSG_PRIORITY selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

TriggerType (MQLONG)
Trigger type.

This controls the conditions under which trigger messages are written as a result of messages arriving on this queue.

It is one of the following:

MQTT_NONE
No trigger messages.

No trigger messages are written as a result of messages on this queue. This has the same effect as setting *TriggerControl* to MQTC_OFF.

MQTT_FIRST
Trigger message when queue depth goes from 0 to 1.

A trigger message is written whenever the queue changes from empty (no messages on the queue) to not-empty (one or more messages on the queue).

MQTT_EVERY
Trigger message for every message.

A trigger message is written every time a message arrives on the queue.

MQTT_DEPTH
Trigger message when depth threshold exceeded.

A trigger message is written when a certain number of messages (*TriggerDepth*) are on the queue. After the trigger message has been written, *TriggerControl* is set to MQTC_OFF to prevent further triggering until it is explicitly turned on again.

To determine the value of this attribute, use the MQIA_TRIGGER_TYPE selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

Usage (MQLONG)
Queue usage.

This indicates what the queue is used for. It is one of the following:

MQUS_NORMAL
Normal usage.

This is a queue that normal applications use when putting and getting messages; the queue is not a transmission queue.

MQUS_TRANSMISSION
Transmission queue.

This is a queue used to hold messages destined for remote queue managers. When a normal application sends a message to a remote queue, the local queue manager stores the message temporarily on the appropriate transmission queue in a special format. A message channel agent then reads the message from the transmission queue, and transports the message to the remote queue manager. For more information about transmission queues, see the *MQSeries Application Programming Guide*.

Only privileged applications can open a transmission queue for MQOO_OUTPUT to put messages on it directly. Only utility applications would normally be expected to do this. Care must be taken that the message data format is correct (see "MQXQH – Transmission queue header" on page 227), otherwise errors may occur during the transmission process. Context is not passed or set unless one of the MQPMO_*_CONTEXT context options is specified.

To determine the value of this attribute, use the MQIA_USAGE selector with the MQINQ call.

Attributes for local definitions of remote queues

The following table summarizes the attributes that are specific to the local definitions of remote queues. The attributes are described in alphabetic order.

Table 63. Attributes for local definitions of remote queues

Attribute	Description	Page
RemoteQMgrName	Name of remote queue manager	364
RemoteQName	Name of remote queue	364
XmitQName	Transmission queue name	365

A local definition of a remote queue is normally used to refer to a queue that exists on a remote queue manager. It specifies the name of the queue manager at which the queue exists, and optionally the name of the transmission queue to be used to convey messages destined for that queue at that queue manager.

However, the same type of definition can also be used for the following purposes:

- Reply queue aliasing

  The name of the definition is the name of a reply-to queue. For more information, see the *MQSeries Intercommunication* book.

- Queue-manager aliasing

  The name of the definition is actually the alias name of a queue manager, not the name of a queue. For more information, see the *MQSeries Intercommunication* book.

RemoteQMgrName (MQCHAR48)
Name of remote queue manager.

The name of the remote queue manager on which the queue *RemoteQName* is defined.

If an application opens the local definition of a remote queue, *RemoteQMgrName* must not be blank and must not be the name of the local queue manager. If *XmitQName* is blank, the local queue whose name is the same as *RemoteQMgrName* is used as the transmission queue. If there is no queue with the name *RemoteQMgrName*, the queue identified by the *DefXmitQName* queue-manager attribute is used.

If this definition is used for a queue-manager alias, *RemoteQMgrName* is the name of the queue manager that is being aliased. It can be the name of the local queue manager. Otherwise, if *XmitQName* is blank when the open occurs, there must be a local queue whose name is the same as *RemoteQMgrName*; this queue is used as the transmission queue.

If this definition is used for a reply-to alias, this name is the name of the queue manager which is to be the *ReplyToQMgr*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

To determine the value of this attribute, use the MQCA_REMOTE_Q_MGR_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_Q_MGR_NAME_LENGTH.

RemoteQName (MQCHAR48)
Name of remote queue.

The name of the queue as it is known on the remote queue manager *RemoteQMgrName*.

If an application opens the local definition of a remote queue, when the open occurs *RemoteQName* must not be blank.

If this definition is used for a queue-manager alias definition, when the open occurs *RemoteQName* must be blank.

If the definition is used for a reply-to alias, this name is the name of the queue that is to be the *ReplyToQ*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

To determine the value of this attribute, use the MQCA_REMOTE_Q_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_Q_NAME_LENGTH.

XmitQName (MQCHAR48)
Transmission queue name.

If this attribute is nonblank when an open occurs, either for a remote queue or for a queue-manager alias definition, it specifies the name of the local transmission queue to be used for forwarding the message.

If *XmitQName* is blank, the local queue whose name is the same as *RemoteQMgrName* is used as the transmission queue. If there is no queue with the name *RemoteQMgrName*, the queue identified by the *DefXmitQName* queue-manager attribute is used.

This attribute is ignored if the definition is being used as a queue-manager alias and *RemoteQMgrName* is the name of the local queue manager.

It is also ignored if the definition is used as a reply-to queue alias definition.

To determine the value of this attribute, use the MQCA_XMIT_Q_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_Q_NAME_LENGTH.

Attributes for alias queues

The following attribute is associated with alias queues:

BaseQName (MQCHAR48)
The queue name to which the alias resolves.

This is the name of a queue that is defined to the local queue manager. (For more information on queue names, see the *MQSeries Application Programming Guide*.) The queue is one of the following types:

MQQT_LOCAL
Local queue.
MQQT_REMOTE
Local definition of a remote queue.

To determine the value of this attribute, use the MQCA_BASE_Q_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_Q_NAME_LENGTH.

Attributes for namelists (MVS/ESA only)

Namelists are supported on MVS/ESA only. The following table summarizes the attributes that are specific to namelists. The attributes are described in alphabetic order.

Table 64. Attributes for namelists

Attribute	Description	Page
NameCount	Number of names in namelist	366
NamelistDesc	Namelist description	366
NamelistName	Namelist name	366
Names	A list of *NameCount* names	367

NameCount (MQLONG)
Number of names in namelist.

This is greater than or equal to zero.

To determine the value of this attribute, use the MQIA_NAME_COUNT selector with the MQINQ call.

NamelistDesc (MQCHAR64)
Namelist description.

This is a field that may be used for descriptive commentary; its value is established by the definition process. The content of the field is of no significance to the queue manager, but the queue manager may require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, this field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters may be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the MQCA_NAMELIST_DESC selector with the MQINQ call.

The length of this attribute is given by MQ_NAMELIST_DESC_LENGTH.

NamelistName (MQCHAR48)
Namelist name.

This is the name of a namelist that is defined on the local queue manager. For more information about namelist names, see the *MQSeries Application Programming Guide*.

Each namelist has a name that is different from the names of other namelists belonging to the queue manager, but may duplicate the names of other queue manager objects of different types (for example, queues).

To determine the value of this attribute, use the MQCA_NAMELIST_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_NAMELIST_NAME_LENGTH.

Names (MQCHAR48×*NameCount*)
A list of *NameCount* names.

Each name is the name of a queue that is defined to the local queue manager. For more information about queue names, see the *MQSeries Application Programming Guide*.

To determine the value of this attribute, use the MQCA_NAMES selector with the MQINQ call.

The length of each name in the list is given by MQ_Q_NAME_LENGTH.

Attributes for process definitions

Process definitions are not supported on: 16-bit Windows, 32-bit Windows. The following table summarizes the attributes that are specific to process definitions. The attributes are described in alphabetic order.

Table 65. Attributes for process definitions

Attribute	Description	Page
ApplId	Application identifier	367
ApplType	Application type	368
EnvData	Environment data	368
ProcessDesc	Process description	369
ProcessName	Process name	369
UserData	User data	369

ApplId (MQCHAR256)
Application identifier.

This is a character string that identifies the application to be started.

This information is for use by a trigger monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

The interpretation to be placed on this information is determined by the trigger-monitor application. For example, *ApplId* could be:

- A program name (for MQAT_MVS applications)
- A CICS transaction ID (for MQAT_CICS applications)

On MVS/ESA, for a CICS application to be started using the CKTI transaction, or an IMS application to be started using the CSQQTRMN application, *ApplId* is a CICS or IMS transaction ID.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_APPL_ID selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_APPL_ID_LENGTH.

ApplType (MQLONG)
Application type.

This identifies the nature of the program to be started in response to the receipt of a trigger message.

This information is for use by a trigger monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

ApplType can have any value, but the following values are recommended for standard types; user-defined application types should be restricted to values in the range MQAT_USER_FIRST through MQAT_USER_LAST:

MQAT_AIX
AIX application (same value as MQAT_UNIX).
MQAT_CICS
CICS transaction.
MQAT_DOS
DOS client application.
MQAT_IMS
IMS application.
MQAT_MVS
MVS or TSO application.
MQAT_NSK
Tandem NSK application.
MQAT_OS2
OS/2 or Presentation Manager application.
MQAT_OS400
OS/400 application.
MQAT_UNIX
UNIX application.
MQAT_WINDOWS
Windows client or 16-bit Windows application.
MQAT_WINDOWS_NT
Windows NT or 32-bit Windows application.
MQAT_USER_FIRST
Lowest value for user-defined application type.
MQAT_USER_LAST
Highest value for user-defined application type.

To determine the value of this attribute, use the MQIA_APPL_TYPE selector with the MQINQ call.

EnvData (MQCHAR128)
Environment data.

This is a character string that contains environment-related information pertaining to the application to be started.

This information is for use by a trigger monitor application that processes messages on the initiation queue; the information is sent to the initiation queue as part of the trigger message.

On MVS/ESA, for a CICS application started using the CKTI transaction, or an IMS application started using the CSQQTRMN transaction, this information is not used.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_ENV_DATA selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_ENV_DATA_LENGTH.

ProcessDesc (MQCHAR64)
Process description.

This is a field that may be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager may require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, the field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters may be translated incorrectly if this field is sent to another queue manager.

To determine the value of this attribute, use the MQCA_PROCESS_DESC selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_DESC_LENGTH.

ProcessName (MQCHAR48)
Process name.

This is the name of a process definition that is defined on the local queue manager.

Each process definition has a name that is different from the names of other process definitions belonging to the queue manager. But the name of the process definition may be the same as the names of other queue manager objects of different types (for example, queues).

To determine the value of this attribute, use the MQCA_PROCESS_NAME selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_NAME_LENGTH.

UserData (MQCHAR128)
User data.

This is a character string that contains user information pertaining to the application to be started.

This information is for use by the trigger monitor application that processes messages on the initiation queue, or the application which is started by the trigger monitor. The information is sent to the initiation queue as part of the trigger message.

The character string cannot contain any nulls. It is padded to the right with blanks if necessary.

To determine the value of this attribute, use the MQCA_USER_DATA selector with the MQINQ call.

The length of this attribute is given by MQ_PROCESS_USER_DATA_LENGTH.

Attributes for the queue manager

The following table summarizes the attributes that are specific to the queue manager. The attributes are described in alphabetic order.

Table 66. Attributes for the queue manager

Attribute	Description	Page
AuthorityEvent	Controls whether authorization (Not Authorized) events are generated	371
ChannelAutoDef	Controls whether automatic channel definition is permitted	371
ChannelAutoDefEvent	Controls whether channel automatic-definition events are generated	371
ChannelAutoDefExit	Name of user exit for automatic channel definition	371
CodedCharSetId	Coded character set identifier	372
CommandInputQName	Command input queue name	372
CommandLevel	Command level	373
DeadLetterQName	Name of dead-letter queue	374
DefXmitQName	Default transmission queue name	375
DistLists	Distribution list support	375
InhibitEvent	Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated	376
LocalEvent	Controls whether local error events are generated	376
MaxHandles	Maximum number of handles	376
MaxMsgLength	Maximum message length in bytes	377
MaxPriority	Maximum priority	377
MaxUncommittedMsgs	Maximum number of uncommitted messages within a unit of work	377
PerformanceEvent	Controls whether performance-related events are generated	378
Platform	Platform on which the queue manager is running	378
QMgrDesc	Queue manager description	379
QMgrName	Queue manager name	379
RemoteEvent	Controls whether remote error events are generated	380
StartStopEvent	Controls whether start and stop events are generated	380
SyncPoint	Syncpoint availability	380
TriggerInterval	Trigger-message interval	380

Some of these attributes are fixed for particular implementations, others can be changed with the ALTER QMGR command. All can be inquired by opening a special MQOT_Q_MGR object, and using the MQINQ call with the handle returned. They can also all be displayed with the DISPLAY QMGR command.

AuthorityEvent (MQLONG)
Controls whether authorization (Not Authorized) events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_AUTHORITY_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute, and the attribute is always in the disabled state.

ChannelAutoDef (MQLONG)
Controls whether automatic channel definition is permitted.

This attribute controls the automatic definition of channels of type MQCHT_RECEIVER and MQCHT_SVRCONN. It is one of the following:

MQCHAD_DISABLED
Channel auto-definition disabled.

MQCHAD_ENABLED
Channel auto-definition enabled.

To determine the value of this attribute, use the MQIA_CHANNEL_AUTO_DEF selector with the MQINQ call.

This attribute is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

ChannelAutoDefEvent (MQLONG)
Controls whether channel automatic-definition events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_CHANNEL_AUTO_DEF_EVENT selector with the MQINQ call.

This attribute is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

ChannelAutoDefExit (MQCHARn)
Name of user exit for automatic channel definition.

If this name is nonblank, and *ChannelAutoDef* has the value MQCHAD_ENABLED, the exit is called each time that the queue manager

is about to create a channel definition. The exit can then do one of the following:

- Allow the creation of the channel definition to proceed without change.
- Modify the attributes of the channel definition that is created.
- Suppress creation of the channel entirely.

Note: Both the length and the value of this attribute are environment specific. See the introduction to the MQCD structure in the MQSeries Intercommunication book for details of the value of this attribute in various environments.

To determine the value of this attribute, use the MQCA_CHANNEL_AUTO_DEF_EXIT selector with the MQINQ call.

This attribute is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

CodedCharSetId (MQLONG)
Coded character set identifier.

This defines the character set used by the queue manager for all character string fields defined in the MQI, including the names of objects, and queue creation date and time. It must be the identifier of a single-byte character set (SBCS). It does not apply to application data carried in the message. The value depends on the environment:

- On MVS/ESA, the value is set from the system parameters when the queue manager is started; the default value is 500. Refer to the *MQSeries for MVS/ESA System Management Guide* for further information.
- On OS/2 and Windows NT, the value is the primary CODEPAGE of the user creating the queue manager.
- On OS/400, the value is that which is set in the environment when the queue manager is first created.
- On OpenVMS, Tandem NSK, and UNIX systems, the value is the default CODESET for the “locale”. of the user creating the queue manager.

To determine the value of this attribute, use the MQIA_CODED_CHAR_SET_ID selector with the MQINQ call.

CommandInputQName (MQCHAR48)
Command input queue name.

This is the name of the command input queue defined on the local queue manager. This is a queue to which applications can send commands, if authorized to do so. The name of the queue depends on the environment:

- On MVS/ESA, the name of the queue is SYSTEM.COMMAND.INPUT, and only MQSC commands can be sent to it. Refer to *MQSeries Command Reference* for details of MQSC commands.
- In all other environments, the name of the queue is SYSTEM.ADMIN.COMMAND.QUEUE, and only PCF commands can be sent to it. However, an MQSC command can be sent to this queue if the MQSC command is enclosed within a PCF command of type

MQCMD_ESCAPE. Refer to *MQSeries Programmable System Management* book for details of PCF commands.

To determine the value of this attribute, use the MQCA_COMMAND_INPUT_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

CommandLevel (MQLONG)
Command Level.

This indicates the level of system control commands supported by the queue manager. The value is one of the following:

MQCMDL_LEVEL_1
Level 1 of system control commands.

This value is returned by the following:

- MQSeries for AIX version 2 release 2
- MQSeries for MVS/ESA:
  - version 1 release 1.1
  - version 1 release 1.2
  - version 1 release 1.3
- MQSeries for OS/2 version 2 release 0
- MQSeries for OS/400:
  - version 2 release 3
  - version 3 release 1
  - version 3 release 6
- MQSeries for Windows version 2 release 0.

MQCMDL_LEVEL_101
MQSeries for Windows version 2 release 0.1.

MQCMDL_LEVEL_110
MQSeries for Windows version 2 release 1.

MQCMDL_LEVEL_114
MQSeries for MVS/ESA version 1 release 1.4.

MQCMDL_LEVEL_120
MQSeries for MVS/ESA version 1 release 2.0.

MQCMDL_LEVEL_200
MQSeries for Windows NT version 2 release 0.

MQCMDL_LEVEL_201
MQSeries for OS/2 version 2 release 0.1.

MQCMDL_LEVEL_220
Level 220 of system control commands.

This value is returned by the following:

- MQSeries for AT&T GIS UNIX version 2 release 2
- MQSeries for SINIX and DC/OSx version 2 release 2
- MQSeries for SunOS version 2 release 2
- MQSeries for Tandem NonStop Kernel version 2 release 2

MQCMDL_LEVEL_221
Level 221 of system control commands.

This value is returned by the following:

- MQSeries for AIX version 2 release 2.1
- MQSeries for Digital OpenVMS version 2 release 2

MQCMDL_LEVEL_320
MQSeries for OS/400 version 3 release 2, and version 3 release 7.

MQCMDL_LEVEL_420
MQSeries for AS/400 version 4 release 2.

MQCMDL_LEVEL_500
Level 500 of system control commands.

This value is returned by the following:

- MQSeries for AIX version 5 release 0
- MQSeries for HP-UX version 5 release 0
- MQSeries for OS/2 version 5 release 0
- MQSeries for Solaris version 5 release 0
- MQSeries for Windows NT version 5 release 0

The set of system control commands that corresponds to a particular value of the *CommandLevel* attribute varies according to the value of the *Platform* attribute; both must be used to decide which system control commands are supported.

To determine the value of this attribute, use the MQIA_COMMAND_LEVEL selector with the MQINQ call.

DeadLetterQName (MQCHAR48)
Name of dead-letter (undelivered-message) queue.

This is the name of a queue defined on the local queue manager. Messages are sent to this queue if they cannot be routed to their correct destination.

For example, messages are put on this queue when:

- A message arrives at a queue manager, destined for a queue that is not yet defined on that queue manager
- A message arrives at a queue manager, but the queue for which it is destined cannot receive it because, possibly:
  - The queue is full
  - Put requests are inhibited
  - The sending node does not have authority to put messages on the queue

Applications can also put messages on the dead-letter queue.

Report messages are treated in the same way as ordinary messages; if the report message cannot be delivered to its destination queue (usually the queue specified by the *ReplyToQ* field in the message descriptor of the original message), the report message is placed on the dead-letter (undelivered-message) queue.

Note: Messages that have passed their expiry time (see the *Expiry* field described in “MQMD – Message descriptor” on page 98) are **not** transferred to this queue when they are discarded. However, an expiration report message (MQRO_EXPIRATION) is still generated and sent to the *ReplyToQ* queue, if requested by the sending application.

Messages are not put on the dead-letter (undelivered-message) queue when the application that issued the put request has been notified synchronously of the problem by means of the reason code returned by the MQPUT or MQPUT1 call (for example, a message put on a local queue for which put requests are inhibited).

Messages on the dead-letter (undelivered-message) queue sometimes have their application message data prefixed with an MQDLH structure. This structure contains extra information that indicates why the message was placed on the dead-letter (undelivered-message) queue. See "MQDLH – Dead-letter header" on page 45 for more details of this structure.

This queue must be a local queue, with a *Usage* attribute of MQUS_NORMAL.

If a dead-letter (undelivered-message) queue is not supported by a queue manager, or one has not been defined, the name is all blanks. All MQSeries queue managers support a dead-letter (undelivered-message) queue, but by default it is not defined.

If the dead-letter (undelivered-message) queue is not defined, or it is full, or unusable for some other reason, a message which would have been transferred to it by a message channel agent is retained instead on the transmission queue.

To determine the value of this attribute, use the MQCA_DEAD_LETTER_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

DefXmitQName (MQCHAR48)
Default transmission queue name.

This is the name of the transmission queue that is used for the transmission of messages to remote queue managers, if there is no other indication of which transmission queue to use.

If there is no default transmission queue, the name is entirely blank. The initial value of this attribute is blank.

To determine the value of this attribute, use the MQCA_DEF_XMIT_Q_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_NAME_LENGTH.

DistLists (MQLONG)
Distribution list support.

This indicates whether the local queue manager supports distribution lists on the MQPUT and MQPUT1 calls. The value is one of the following:

MQDL_SUPPORTED
Distribution lists supported.

MQDL_NOT_SUPPORTED
Distribution lists not supported.

To determine the value of this attribute, use the MQIA_DIST_LISTS selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

This attribute is supported in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

InhibitEvent (MQLONG)
Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_INHIBIT_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

LocalEvent (MQLONG)
Controls whether local error events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_LOCAL_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

MaxHandles (MQLONG)
Maximum number of handles.

This is the maximum number of open handles that any one task can use concurrently. Each successful MQOPEN call for a single queue (or for an object that is not a queue) uses one handle. That handle becomes available for reuse when the object is closed. However, when a distribution list is opened, each queue in the distribution list is allocated a separate handle, and so that MQOPEN call uses as many handles as there are queues in the distribution list. This must be taken into account when deciding on a suitable value for *MaxHandles*.

The MQPUT1 call performs an MQOPEN call as part of its processing; as a result, MQPUT1 uses as many handles as MQOPEN would, but the handles are used only for the duration of the MQPUT1 call itself.

On MVS/ESA, “task” means a CICS task, an MVS task, or an IMS dependent region.

The value is in the range 1 through 999 999 999. The default value is determined by the environment:

- On MVS/ESA, the default value is 100.
- In all other environments, the default value is 256.

To determine the value of this attribute, use the MQIA_MAX_HANDLES selector with the MQINQ call.

MaxMsgLength (MQLONG)
Maximum message length in bytes.

This is the length of the longest physical message that can be handled by the queue manager. The *MaxMsgLength* queue-manager attribute can be set independently of the *MaxMsgLength* local-queue attribute, and the longest physical message that can be placed on a queue is the lesser of those two values.

If the queue manager supports segmentation, it is possible for an application to put a message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MQMF_SEGMENTATION_ALLOWED flag. In these circumstances, the longest message that can be put depends on resource constraints imposed by the operating system or by the environment in which the application is running.

The lower limit for this attribute is 32 KB (32 768 bytes). The upper limit is determined by the environment:

- On AIX, HP-UX, OS/2, Sun Solaris, and Windows NT, the maximum message length is 100 MB (104 857 600 bytes).
- On OpenVMS, MVS/ESA, OS/400, Tandem NSK, UNIX systems not listed above, 16-bit Windows, and 32-bit Windows, the maximum message length is 4 MB (4 194 304 bytes).

To determine the value of this attribute, use the MQIA_MAX_MSG_LENGTH selector with the MQINQ call.

MaxPriority (MQLONG)
Maximum priority.

This is the maximum message priority supported by the queue manager. Priorities range from zero (lowest) to *MaxPriority* (highest).

To determine the value of this attribute, use the MQIA_MAX_PRIORITY selector with the MQINQ call.

MaxUncommittedMsgs (MQLONG)
Maximum number of uncommitted messages within a unit of work.

This is the maximum number of uncommitted messages that can exist within a unit of work. The number of uncommitted messages is the sum of the following since the start of the current unit of work:

- Messages put by the application with the MQPMO_SYNCPOINT option
- Messages retrieved by the application with the MQGMO_SYNCPOINT option
- Trigger messages and COA report messages generated by the queue manager for messages put with the MQPMO_SYNCPOINT option

- COD report messages generated by the queue manager for messages retrieved with the MQGMO_SYNCPOINT option

The following are *not* counted as uncommitted messages:

- Messages put or retrieved by the application outside a unit of work
- Trigger messages or COA/COD report messages generated by the queue manager as a result of messages put or retrieved outside a unit of work.
- Expiration report messages generated by the queue manager (even if the call causing the expiration report message specified MQGMO_SYNCPOINT)
- Event messages generated by the queue manager (even if the call causing the event message specified MQPMO_SYNCPOINT or MQGMO_SYNCPOINT)

Note: Exception report messages are generated by the Message Channel Agent (MCA), or by the application, and so are treated in the same way as ordinary messages put or retrieved by the application.

The lower limit for this attribute is 1; the upper limit is 999 999 999.

To determine the value of this attribute, use the MQIA_MAX_UNCOMMITTED_MSGS selector with the MQINQ call.

On MVS/ESA, this attribute is not supported.

PerformanceEvent (MQLONG)
Controls whether performance-related events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_PERFORMANCE_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

Platform (MQLONG)
Platform on which the queue manager is running.

This indicates the architecture of the platform on which the queue manager is running:

MQPL_AIX
AIX (same value as MQPL_UNIX).

MQPL_MVS
MVS/ESA.

MQPL_NSK
Tandem NonStop Kernel.

MQPL_OS2
OS/2.

MQPL_OS400
OS/400.

MQPL_UNIX
UNIX systems.

MQPL_VMS
OpenVMS.

MQPL_WINDOWS
16-bit Windows.

MQPL_WINDOWS_NT
Windows NT or 32-bit Windows.

To determine the value of this attribute, use the MQIA_PLATFORM selector with the MQINQ call.

QMgrDesc (MQCHAR64)
Queue manager description.

This is a field that may be used for descriptive commentary. The content of the field is of no significance to the queue manager, but the queue manager may require that the field contain only characters that can be displayed. It cannot contain any null characters; if necessary, it is padded to the right with blanks. In a DBCS installation, this field can contain DBCS characters (subject to a maximum field length of 64 bytes).

Note: If this field contains characters that are not in the queue manager's character set (as defined by the *CodedCharSetId* queue manager attribute), those characters may be translated incorrectly if this field is sent to another queue manager.

- On MVS/ESA, the default value is:

  ```
  MQSeries for MVS/ESA Vx.y.z
  ```

  where x, y, and z are replaced by the version, release, and modification numbers, respectively.

- In all other environments, the default value is blanks.

To determine the value of this attribute, use the MQCA_Q_MGR_DESC selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_DESC_LENGTH.

QMgrName (MQCHAR48)
Queue manager name.

This is the name of the local queue manager, that is, the name of the queue manager to which the application is connected.

The first 12 characters of the name are used to construct a unique message identifier (see the *MsgId* field described in "MQMD – Message descriptor" on page 98). Queue managers that can intercommunicate must therefore have names that differ in the first 12 characters, in order for message identifiers to be unique in the queue-manager network.

On MVS/ESA, the name is the same as the subsystem name, which is limited to 4 nonblank characters.

To determine the value of this attribute, use the MQCA_Q_MGR_NAME selector with the MQINQ call. The length of this attribute is given by MQ_Q_MGR_NAME_LENGTH.

RemoteEvent (MQLONG)
Controls whether remote error events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_REMOTE_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

StartStopEvent (MQLONG)
Controls whether start and stop events are generated.

It is one of the following:

MQEVR_DISABLED
Event reporting disabled.

MQEVR_ENABLED
Event reporting enabled.

To determine the value of this attribute, use the MQIA_START_STOP_EVENT selector with the MQINQ call.

On MVS/ESA, the MQINQ call cannot be used to determine the value of this attribute.

SyncPoint (MQLONG)
Syncpoint availability.

This indicates whether the local queue manager supports units of work and syncpointing with the MQGET, MQPUT, and MQPUT1 calls.

MQSP_AVAILABLE
Units of work and syncpointing available.

MQSP_NOT_AVAILABLE
Units of work and syncpointing not available.

On MVS/ESA and OS/400, this value is never returned.

To determine the value of this attribute, use the MQIA_SYNCPOINT selector with the MQINQ call.

TriggerInterval (MQLONG)
Trigger-message interval.

This is a time interval (in milliseconds) used to restrict the number of trigger messages. This is relevant only when the *TriggerType* is MQTT_FIRST. In this case trigger messages are normally generated only when a suitable message arrives on the queue, and the queue was previously empty. Under certain circumstances, however, an additional

trigger message can be generated with MQTT_FIRST triggering even if the queue was not empty. These additional trigger messages are not generated more often than every *TriggerInterval* milliseconds.

For more information on triggering, see the *MQSeries Application Programming Guide*.

The value is not less than 0 and not greater than 999 999 999. The default value is 999 999 999.

To determine the value of this attribute, use the MQIA_TRIGGER_INTERVAL selector with the MQINQ call.

This attribute is not supported in the following environments: 16-bit Windows, 32-bit Windows.

Chapter 5. Return codes

This book contains the return codes associated with the API. The return codes associated with Programmable Command Format (PCF) commands are listed in the *MQSeries Programmable System Management* book.

For each call, a completion code and a reason code are returned by the queue manager or by an exit routine, to indicate the success or failure of the call.

Applications must not depend upon errors being checked for in a specific order, except where specifically noted. If more than one completion code or reason code could arise from a call, the particular error reported depends on the implementation.

Completion code

The completion code parameter (*CompCode*) allows the caller to see quickly whether the call completed successfully, completed partially, or failed.

The following is a list of completion codes, with more detail than is given in the call descriptions:

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.

MQCC_WARNING
: Warning (partial completion).

  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 about the partial completion.

MQCC_FAILED
: Call failed.

  The processing of the call did not complete, and the state of the queue manager is normally unchanged; exceptions are specifically noted. The *CompCode* and *Reason* output parameters have been set; other parameters are unchanged, except where noted.

  The reason may be a fault in the application program, or it may be a result of some situation external to the program, for example the application's authority may have been revoked. The *Reason* parameter gives additional information about the error.

Reason code

The reason code parameter (*Reason*) is a qualification to the completion code parameter (*CompCode*).

If there is no special reason to report, MQRC_NONE is returned. A successful call returns MQCC_OK and MQRC_NONE.

If the completion code is either MQCC_WARNING or MQCC_FAILED, the queue manager always reports a qualifying reason; details are given under each call description.

Where user exit routines set completion codes and reasons, they should adhere to these rules.

Any special reason values defined by user exits should be less than zero, to ensure that they do not conflict with values defined by the queue manager. Exits can set reasons already defined by the queue manager, where these are appropriate.

Reason codes also occur in:

- The *Reason* field of the MQDLH structure (for messages on the dead-letter queue)
- The *Feedback* field of the MQMD structure (message descriptor)

The following is a list of reason codes, in alphabetic order, with more detail than is given in the call descriptions. See "MQRC_* (Reason code)" on page 470 for a list of reason codes in numeric order.

MQRC_ADAPTER_CONN_LOAD_ERROR
(2129, X'851') Unable to load adapter connection module.

On an MQCONN call, the connection handling module (CSQBCON for batch and CSQQCONN for IMS) could not be loaded, so the adapter could not link to it.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the correct library concatenation has been specified in the batch application program execution JCL, and in the MQSeries startup JCL.

MQRC_ADAPTER_CONV_LOAD_ERROR
(2133, X'855') Unable to load data conversion services modules.

On an MQGET call, the adapter (batch or IMS) could not load the data conversion services modules.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the correct library concatenation has been specified in the batch application program execution JCL, and in the MQSeries startup JCL.

MQRC_ADAPTER_DEFS_ERROR
(2131, X'853') Adapter subsystem definition module not valid.

On an MQCONN call, the subsystem definition module (CSQBDEFV for batch and CSQQDEFV for IMS) does not contain the required control block identifier.

This reason code occurs only on MVS/ESA.

Corrective action: Check your library concatenation. If this is correct, check that the CSQBDEFV or CSQQDEFV module contains the required subsystem ID.

MQRC_ADAPTER_DEFS_LOAD_ERROR
(2132, X'854') Unable to load adapter subsystem definition module.

On an MQCONN call, the subsystem definition module (CSQBDEFV for batch and CSQQDEFV for IMS) could not be loaded.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the correct library concatenation has been specified in the application program execution JCL, and in the MQSeries startup JCL.

MQRC_ADAPTER_DISC_LOAD_ERROR
(2138, X'85A') Unable to load adapter disconnection module.

On an MQDISC call, the disconnect handling module (CSQBDSC for batch and CSQQDISC for IMS) could not be loaded, so the adapter could not link to it.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the correct library concatenation has been specified in the application program execution JCL, and in the MQSeries startup JCL.

Applications should ensure that any uncommitted updates are backed out. Any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_ADAPTER_NOT_AVAILABLE
(2204, X'89C') Adapter not available.

This is issued only for CICS applications, if any call is issued and the CICS adapter (a Task Related User Exit) has been disabled, or has not been enabled.

This reason code occurs only on MVS/ESA.

Corrective action: The application should tidy up and terminate.

Applications should ensure that any uncommitted updates are backed out. Any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_ADAPTER_SERV_LOAD_ERROR
(2130, X'852') Unable to load adapter service module.

On an API call, the batch adapter could not load the API service module CSQBSRV, and so could not link to it.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the correct library concatenation has been specified in the batch application program execution JCL, and in the MQSeries startup JCL.

MQRC_ADAPTER_STORAGE_SHORTAGE
(2127, X'84F') Insufficient storage for adapter.

On an MQCONN call, the adapter was unable to acquire storage.

This reason code occurs only on MVS/ESA.

Corrective action: Notify the system programmer.

The system programmer should determine why the system is short on storage, and take appropriate action, for example, increase the region size on the step or job card.

MQRC_ALIAS_BASE_Q_TYPE_ERROR
(2001, X'7D1') Alias base queue not a valid type.

An MQOPEN or MQPUT1 call was issued specifying an alias queue as the destination, but the *BaseQName* in the alias queue definition resolves to a queue that is not a local queue, or local definition of a remote queue.

Corrective action: Correct the queue definitions.

MQRC_ALREADY_CONNECTED
(2002, X'7D2') Application already connected.

An MQCONN call was issued, but the application is already connected to the queue manager.

On MVS/ESA, this reason code occurs for batch and IMS applications only; it does not occur for CICS applications.

Corrective action: None. The *Hconn* parameter returned has the same value as was returned for the previous MQCONN call.

Note: An MQCONN call that returns this reason code does *not* mean that an additional MQDISC call must be issued in order to disconnect from the queue manager. If this reason code is returned because the application (or portion thereof) has been called in a situation where the connect has already been done, a corresponding MQDISC should *not* be issued, because this will cause the application that issued the original MQCONN call to be disconnected as well.

MQRC_ANOTHER_Q_MGR_CONNECTED
(2103, X'837') Another queue manager already connected.

An MQCONN call was issued, but the thread or process is already connected to a different queue manager. The thread or process can connect to only one queue manager at a time.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Use the MQDISC call to disconnect from the queue manager which is already connected, and then issue the MQCONN call to connect to the new queue manager.

Note: Disconnecting from the existing queue manager will close any queues which are currently open; it is recommended that any uncommitted units of work should be committed or backed out before the MQDISC call is used.

MQRC_API_EXIT_LOAD_ERROR
(2183, X'887') Unable to load API crossing exit.

The API crossing exit module could not be linked.

If this reason is returned when the API crossing exit is invoked *after* the call has been executed, the call itself may have executed correctly.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the correct library concatenation has been specified, and that the API crossing exit module is executable and correctly named.

Applications should ensure that any uncommitted updates are backed out. Any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_ASID_MISMATCH
(2157, X'86D') Primary and home ASIDs differ.

On any API call, the caller's primary ASID was found to be different from the home ASID.

This reason code occurs only on MVS/ESA.

Corrective action: Correct the application. MQM calls cannot be issued in cross-memory mode.

Applications should ensure that any uncommitted updates are backed out. Any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_BACKED_OUT
(2003, X'7D3') Unit of work encountered fatal error or backed out.

This occurs in the following cases:

- On an MQCMIT or MQDISC call, when the commit operation has failed and the unit of work has been backed out. All protected resources have been returned to their state at the start of the unit of work. The MQCMIT call returns completion code MQCC_FAILED; the MQDISC call returns completion code MQCC_WARNING.

  On MVS/ESA, this reason code occurs only for batch applications.

- On an MQGET, MQPUT, or MQPUT1 call that is operating within a unit of work, when the unit of work has already encountered an error that prevents the unit of work being committed (for example, when the log space is exhausted). The application must issue the appropriate call to back out the unit of work. For a unit of work coordinated by the queue manager, this call is the MQBACK call, although the MQCMIT call has the same effect in these circumstances.

  On MVS/ESA this case does not occur.

On OS/400, this reason code does not occur.

Corrective action: Check the returns from previous calls to the queue manager. For example, a previous MQPUT call may have failed.

MQRC_BO_ERROR
(2134, X'856') Begin-options structure not valid.

On an MQBEGIN call, the begin-options structure MQBO is not valid, for one of the following reasons:

- The *StrucId* mnemonic eye-catcher is not MQBO_STRUC_ID.
- The *Version* field is not MQBO_VERSION_1.
- The parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- The queue manager cannot copy the changed structure to application storage, even though the call is successful. This can occur, for example, if the pointer points to read-only storage.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQBO structure. Ensure that required input fields are set correctly.

MQRC_BRIDGE_STARTED
(2125, X'84D') Bridge started.

The IMS bridge has been started.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_BRIDGE_STOPPED
(2126, X'84E') Bridge stopped.

The IMS bridge has been stopped.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_BUFFER_ERROR
(2004, X'7D4') Buffer parameter not valid.

Buffer is not valid. The parameter pointer is not valid, or points to read-only storage for MQGET calls, or to storage that cannot be accessed for the entire length specified by *BufferLength*. (It is not always possible to detect parameter pointers that are not valid; if it is not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

MQRC_BUFFER_LENGTH_ERROR
(2005, X'7D5') Buffer length parameter not valid.

BufferLength or the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

This reason can also be returned to an MQ client program on the MQCONN call if the negotiated maximum message size for the channel is smaller than the fixed part of any call structure.

Corrective action: Specify a nonnegative value.

MQRC_CALL_IN_PROGRESS
(2219, X'8AB') MQI call reentered before previous call complete.

The application issued an MQI call whilst another MQI call was already being processed for that connection. Only one call per application connection can be processed at a time.

Concurrent calls can arise only in certain specialized situations, such as in an exit invoked as part of the processing of an MQI call. For example, the data-conversion exit may be invoked as part of the processing of the MQGET call.

- On MVS/ESA, concurrent calls can arise only with batch or IMS applications; an example is when a subtask ends while an MQI call is in progress (for example, an MQGET which is waiting), and there is an end-of-task exit routine that issues another MQI call.

- On OS/2, Windows client, and Windows NT concurrent calls can also arise if an MQI call is issued in response to a user message while another MQI call is in progress.

Corrective action: Ensure that an MQI call cannot be issued while another one is active. Do not issue MQI calls from within a data-conversion exit.

On MVS/ESA, if you want to provide a subtask to allow an application that is waiting for a message to arrive to be canceled, use MQGET with MQGMO_SET_SIGNAL, rather than with MQGMO_WAIT, to wait for the message.

MQRC_CFH_ERROR
(2235, X'8BB') PCF header structure not valid.

On an MQPUT or MQPUT1 call, the PCF header structure MQCFH in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQCFH structure. Ensure that the fields are set correctly.

MQRC_CFIL_ERROR
(2236, X'8BC') PCF integer list parameter structure not valid.

On an MQPUT or MQPUT1 call, the PCF integer list parameter structure MQCFIL in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQCFIL structure. Ensure that the fields are set correctly.

MQRC_CFIN_ERROR
(2237, X'8BD') PCF integer parameter structure not valid.

On an MQPUT or MQPUT1 call, the PCF integer parameter structure MQCFIN in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQCFIN structure. Ensure that the fields are set correctly.

MQRC_CFSL_ERROR
(2238, X'8BE') PCF string list parameter structure not valid.

On an MQPUT or MQPUT1 call, the PCF string list parameter structure MQCFSL in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQCFSL structure. Ensure that the fields are set correctly.

MQRC_CFST_ERROR
(2239, X'8BF') PCF string parameter structure not valid.

On an MQPUT or MQPUT1 call, the PCF string parameter structure MQCFST in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQCFST structure. Ensure that the fields are set correctly.

MQRC_CHANNEL_ACTIVATED
(2295, X'8F7') Channel activated.

This condition is detected when a channel which has been waiting to become active, and for which a Channel Not Activated event has been generated, is now able to become active because an active slot has been released by another channel.

This event is not generated for a channel which is able to become active without waiting for an active slot to be released.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_CHANNEL_AUTO_DEF_ERROR
(2234, X'8BA') Automatic channel definition failed.

This condition is detected when the automatic definition of a channel fails; this may be because an error occurred during the definition process, or because the channel automatic-definition exit inhibited the definition. Additional information is returned in the event message indicating the reason for the failure.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Examine the additional information returned in the event message to determine the reason for the failure.

MQRC_CHANNEL_AUTO_DEF_OK
(2233, X'8B9') Automatic channel definition succeeded.

This condition is detected when the automatic definition of a channel is successful. The channel is defined by the MCA.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_CHANNEL_CONV_ERROR
(2284, X'8EC') Channel conversion error.

This condition is detected when a channel is unable to do data conversion and the MQGET call to get a message from the transmission queue resulted in a data conversion error. The conversion reason code identifies the reason for the failure.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_CHANNEL_NOT_ACTIVATED
(2296, X'8F8') Channel cannot be activated.

This condition is detected when a channel is required to become active, either because it is starting or because it is about to make another attempt to establish connection with its partner. However, it is unable to do so because

the limit on the number of active channels has been reached (see the MaxActiveChannels parameter in the qm.ini file, or, for MVS/ESA see the ACTCHL parameter in CSQXPARM). The channel waits until it is able to take over an active slot released when another channel ceases to be active. At that time a Channel Activated event is generated.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_CHANNEL_STARTED
(2282, X'8EA') Channel started.

One of the following has occurred:

- An operator has issued a Start Channel command.
- An instance of a channel has been successfully established.

  This condition is detected when Initial Data negotiation is complete and resynchronization has been performed where necessary such that message transfer can proceed.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_CHANNEL_STOPPED
(2283, X'8EB') Channel stopped.

This condition is detected when the channel has been stopped. The reason qualifier identifies the reasons for stopping.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_CHAR_ATTR_LENGTH_ERROR
(2006, X'7D6') Length of character attributes not valid.

CharAttrLength is negative (for MQINQ or MQSET calls), or is not large enough to hold all selected attributes (MQSET calls only). This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a value large enough to hold the concatenated strings for all selected attributes.

MQRC_CHAR_ATTRS_ERROR
(2007, X'7D7') 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 that are not valid; if it is not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

MQRC_CHAR_ATTRS_TOO_SHORT
(2008, X'7D8') 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 filled in with as many character attributes as there is room for. Only complete attribute strings

are returned: if there is insufficient space remaining to accommodate an attribute in its entirety, that attribute and subsequent character attributes are omitted. Any space at the end of the string not used to hold an attribute is unchanged.

An attribute that represents a set of values (for example, the *Names* attribute) is treated as a single entity—either all of its values are returned, or none.

Corrective action: Specify a large enough value, unless only a subset of the values is needed.

MQRC_CICS_BRIDGE_RESTRICTION
(2187, X'88B') Requested function not supported by CICS bridge.

It is not permitted to use the MQI from user transactions that are run in an MQSeries-CICS bridge environment where the bridge exit also uses the MQI. The MQI request fails. If this occurs in the bridge exit, it will result in a transaction abend. If it occurs in the user transaction, this may result in a transaction abend.

This reason code occurs only on MVS/ESA.

Corrective action: The transaction cannot be run using the MQSeries-CICS bridge. Refer to the appropriate CICS manual for information about restrictions in the MQSeries-CICS bridge environment.

MQRC_CICS_WAIT_FAILED
(2140, X'85C') Wait request rejected by CICS.

On any API call, the CICS adapter issued an EXEC CICS WAIT request, but the request was rejected by CICS.

This reason code occurs only on MVS/ESA.

Corrective action: Examine the CICS trace data for actual response codes. The most likely cause is that the task has been canceled by the operator or by the system.

MQRC_CNO_ERROR
(2139, X'85B') Connect-options structure not valid.

On an MQCONNX call, the connect-options structure MQCNO is not valid, for one of the following reasons:

- The *StrucId* mnemonic eye-catcher is not MQCNO_STRUC_ID.
- The *Version* field is not MQCNO_VERSION_1.
- The parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- The queue manager cannot copy the changed structure to application storage, even though the call is successful. This can occur, for example, if the parameter pointer points to read-only storage.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQCNO structure. Ensure that required input fields are set correctly.

MQRC_COD_NOT_VALID_FOR_XCF_Q
(2106, X'83A') COD report option not valid for XCF queue.

An MQPUT or MQPUT1 call was issued, but the *Report* field in the message

descriptor MQMD specifies one of the MQRO_COD_★ options and the target queue is an XCF queue. MQRO_COD_★ options cannot be specified for XCF queues.

This reason code occurs only on MVS/ESA.

Corrective action: Remove the relevant MQRO_COD_★ option.

MQRC_CONN_ID_IN_USE
(2160, X'870') Connection identifier already in use.

On an MQCONN call, the connection identifier assigned by MQSeries to the connection between a CICS or IMS allied address space and the queue manager conflicts with the connection identifier of another connected CICS or IMS system. The connection identifier assigned is as follows:

- For CICS, the applid
- For IMS, the IMSID parameter on the IMSCTRL (sysgen) macro, or the IMSID parameter on the execution parameter (EXEC card in IMS control region JCL)
- For batch, the job name
- For TSO, the user ID

A conflict arises only if there are two CICS systems, two IMS systems, or one each of CICS and IMS, having the same connection identifiers. Batch and TSO connections need not have unique identifiers.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the naming conventions used in different systems that might connect to MQSeries do not conflict.

MQRC_CONNECTION_BROKEN
(2009, X'7D9') Connection to queue manager lost.

Connection to the queue manager has been lost. This can occur because the queue manager has ended. If the call is an MQGET call with the MQGMO_WAIT option, the wait has been canceled.

If this reason occurs with MQCONN, the queue manager may have been stopped and restarted, and now be available again. All previous handles are now invalid, but the application can attempt to reestablish connection by issuing MQCONN again.

Note that for MQ client applications it is possible that the call did complete successfully, even though this reason code is returned with a *CompCode* of MQCC_FAILED.

Corrective action: Applications can attempt to reestablish connection by issuing the MQCONN call. It may be necessary to poll until a successful response is received.

On MVS/ESA, for CICS applications, it is not necessary to issue the MQCONN call, because CICS applications are connected automatically.

Applications should ensure that any uncommitted updates are backed out. Any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_CONNECTION_NOT_AUTHORIZED
(2217, X'8A9') Not authorized for connection.

This reason code arises only for CICS applications. For these, connection to the queue manager is done by the adapter. If that connection fails because the CICS subsystem is not authorized to connect to the queue manager, this reason code is issued whenever an application running under that subsystem subsequently issues an MQI call.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the subsystem is authorized to connect to the queue manager.

MQRC_CONNECTION_QUIESCING
(2202, X'89A') Connection quiescing.

This occurs only for CICS and IMS applications.

It is issued if the connection to the queue manager is in quiescing state, and an application attempts to connect to the queue manager, either with MQCONN or by attempting to open a queue when no connection is established.

It is also issued if the connection to the queue manager is in quiescing state, and an application issues one of the following calls:

- MQOPEN, with MQOO_FAIL_IF_QUIESCING included in the *Options* parameter
- MQGET, with MQGMO_FAIL_IF_QUIESCING included in the *Options* field of the *GetMsgOpts* parameter
- MQPUT or MQPUT1, with MQPMO_FAIL_IF_QUIESCING included in the *Options* field of the *PutMsgOpts* parameter

This reason code occurs only on MVS/ESA.

Corrective action: The application should tidy up and terminate.

MQRC_CONNECTION_STOPPING
(2203, X'89B') Connection shutting down.

This is issued only for CICS and IMS applications, if any call is issued when the connection to the queue manager is shutting down. If the call is an MQGET call with the MQGMO_WAIT option, the wait has been canceled. No more message-queuing calls can be issued.

Note that the MQRC_CONNECTION_BROKEN reason may be returned instead if, as a result of system scheduling factors, the queue manager shuts down before the call completes.

This reason code occurs only on MVS/ESA.

Corrective action: The application should tidy up and terminate.

Applications should ensure that any uncommitted updates are backed out. Any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_CONTEXT_HANDLE_ERROR
(2097, X'831') Queue handle referred to does not save context.

On an MQPUT or MQPUT1 call, MQPMO_PASS_IDENTITY_CONTEXT or MQPMO_PASS_ALL_CONTEXT was specified, but the handle specified in

the *Context* field of the *PutMsgOpts* parameter is either not a valid queue handle, or it is a valid queue handle but the queue was not opened with MQOO_SAVE_ALL_CONTEXT.

Corrective action: Specify MQOO_SAVE_ALL_CONTEXT when the queue referred to is opened.

MQRC_CONTEXT_NOT_AVAILABLE
(2098, X'832') Context not available for queue handle referred to.

On an MQPUT or MQPUT1 call, MQPMO_PASS_IDENTITY_CONTEXT or MQPMO_PASS_ALL_CONTEXT was specified, but the queue handle specified in the *Context* field of the *PutMsgOpts* parameter has no context associated with it. This arises if no message has yet been successfully retrieved with the queue handle referred to, or if the last successful MQGET call was a browse.

This condition does not arise if the message that was last retrieved had no context associated with it.

On MVS/ESA, if a message is received by a message channel agent which is putting messages with the authority of the user identifier in the message, this code is returned in the *Feedback* field of an exception report if the message has no context associated with it.

Corrective action: Ensure that a successful nonbrowse get call has been issued with the queue handle referred to.

MQRC_CONVERTED_MSG_TOO_BIG
(2120, X'848') Converted message too big for application buffer.

On an MQGET call, with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the message data expanded during data conversion and exceeded the size of the buffer provided by the application. However, the message had already been removed from the queue because prior to conversion the message data could be accommodated in the application buffer without truncation.

To avoid data being lost, the message is returned unconverted, with the *CompCode* parameter of the MQGET call set to MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

This reason can also occur on the MQXCNVC call, when the *TargetBuffer* parameter is too small too accommodate the converted string, and the string has been truncated to fit in the buffer. The length of valid data returned is given by the *DataLength* parameter; in the case of a DBCS string or mixed SBCS/DBCS string, this length may be *less than* the length of *TargetBuffer*.

Corrective action: For the MQGET call, check that the exit is converting the message data correctly and setting the output length *DataLength* to the appropriate value. If it is, the application issuing the MQGET call must provide a larger buffer for the *Buffer* parameter.

For the MQXCNVC call, if the string must be converted without truncation, provide a larger output buffer.

MQRC_DATA_LENGTH_ERROR
(2010, X'7DA') 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 an invalid parameter pointer; if not detected, unpredictable results occur.)

This reason can also be returned to an MQ client program on the MQGET, MQPUT, or MQPUT1 call, if the application message data is longer than the negotiated maximum message size for the channel.

Corrective action: Correct the parameter.

If the error occurs for an MQ client program, also check that the maximum message size for the channel is big enough to accommodate the message being sent; if it is not big enough, increase the maximum message size for the channel.

MQRC_DBCS_ERROR
(2150, X'866') DBCS string not valid.

On the MQXCNVC call, the *SourceCCSID* parameter specifies the coded character-set identifier of a double-byte character set (DBCS), but the *SourceBuffer* parameter does not contain a valid DBCS string. This may be because the string contains characters which are not valid DBCS characters, or because the string is a mixed SBCS/DBCS string and the shift-out/shift-in characters are not correctly paired.

Corrective action: Specify a valid string.

MQRC_DEF_XMIT_Q_TYPE_ERROR
(2198, X'896') Default transmission queue not local.

An MQOPEN or MQPUT1 call was issued specifying a remote queue as the destination. Either a local definition of the remote queue was specified, or a queue-manager alias was being resolved, but in either case the *XmitQName* attribute in the local definition is blank.

Because there is no transmission queue defined with the same name as the destination queue manager, the local queue manager has attempted to use the default transmission queue. However, although there is a queue defined by the *DefXmitQName* queue-manager attribute, it is not a local queue.

Corrective Action: Do one of the following:

- Specify a local transmission queue as the value of the *XmitQName* attribute in the local definition of the remote queue.
- Define a local transmission queue with a name which is the same as that of the remote queue manager.
- Specify a local transmission queue as the value of the *DefXmitQName* queue-manager attribute.

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

MQRC_DEF_XMIT_Q_USAGE_ERROR
(2199, X'897') Default transmission queue usage error.

An MQOPEN or MQPUT1 call was issued specifying a remote queue as the destination. Either a local definition of the remote queue was specified, or a queue-manager alias was being resolved, but in either case the *XmitQName* attribute in the local definition is blank.

Because there is no transmission queue defined with the same name as the destination queue manager, the local queue manager has attempted to use the default transmission queue. However, the queue defined by the *DefXmitQName* queue-manager attribute does not have a *Usage* attribute of MQUS_TRANSMISSION.

Corrective Action: Do one of the following:

- Specify a local transmission queue as the value of the *XmitQName* attribute in the local definition of the remote queue.
- Define a local transmission queue with a name which is the same as that of the remote queue manager.
- Specify a different local transmission queue as the value of the *DefXmitQName* queue-manager attribute.
- Change the *Usage* attribute of the *DefXmitQName* queue to MQUS_TRANSMISSION.

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

MQRC_DEST_ENV_ERROR
(2263, X'8D7') Destination environment data error.

This reason occurs when a channel exit that processes reference messages detects an error in the destination environment data of a reference message header (MQRMH). One of the following is true:

- *DestEnvLength* is less than zero.
- Destination environment data is not present although *DestEnvLength* is greater than zero.
- The range defined by *DestEnvOffset* and *DestEnvLength* is not wholly beyond the fixed fields in the MQRMH structure and within *StrucLength* bytes from the start of the structure.

The exit returns this reason in the *Feedback* field of the MQCXP structure. If an exception report is requested, it is copied to the *Feedback* field of the MQMD associated with the report.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify the destination environment data correctly.

MQRC_DEST_NAME_ERROR
(2264, X'8D8') Destination name data error.

This reason occurs when a channel exit that processes reference messages detects an error in the destination name data of a reference message header (MQRMH). One of the following is true:

- *DestNameLength* is less than zero.
- Destination name data is not present although *DestNameLength* is greater than zero.
- The range defined by *DestNameOffset* and *DestNameLength* is not wholly beyond the fixed fields in the MQRMH structure and within *StrucLength* bytes from the start of the structure.

The exit returns this reason in the *Feedback* field of the MQCXP structure. If an exception report is requested, it is copied to the *Feedback* field of the MQMD associated with the report.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify the destination name data correctly.

MQRC_DH_ERROR
(2135, X'857') Distribution header structure not valid.

On an MQPUT or MQPUT1 call, the distribution header structure MQDH in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQDH structure. Ensure that the fields are set correctly.

MQRC_DLH_ERROR
(2141, X'85D') Dead letter header structure not valid.

On an MQPUT or MQPUT1 call, the dead letter header structure MQDLH in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQDLH structure. Ensure that the fields are set correctly.

MQRC_DUPLICATE_RECOV_COORD
(2163, X'873') Recovery coordinator already exists.

On an MQCONN call, a recovery coordinator already exists for the connection name specified on the connection call issued by the adapter.

A conflict arises only if there are two CICS systems, two IMS systems, or one each of CICS and IMS, having the same connection identifiers. Batch and TSO connections need not have unique identifiers.

This reason code occurs only on MVS/ESA.

Corrective action: Ensure that the naming conventions used in different systems that might connect to MQSeries do not conflict.

MQRC_DYNAMIC_Q_NAME_ERROR
(2011, X'7DB') Name of dynamic queue not valid.

On the MQOPEN call, a model queue is specified in the *ObjectName* field of the *ObjDesc* parameter, but the *DynamicQName* field is not valid, for one of the following reasons:

- Characters are present that are not valid for a queue name.
- An asterisk is present beyond the 33rd position (and before any null character).
- An asterisk is present followed by characters which are not null and not blank.

Corrective action: Specify a valid name.

MQRC_ENVIRONMENT_ERROR
(2012, X'7DC') Call not valid in environment.

The call is not valid for the current environment.

- On MVS/ESA, the MQCMIT and MQBACK calls cannot be issued in the CICS or IMS environment.
- On OpenVMS, OS/2, Tandem NSK, UNIX systems, and Windows NT, one of the following applies:
  - The application is linked to the wrong libraries (threaded or nonthreaded).
  - An MQBEGIN, MQCMIT, or MQBACK call was issued, but an external unit-of-work manager is in use or the queue manager does not support units of work.
  - The MQBEGIN call was issued in an MQ client environment.
- On OS/400, this reason code does not occur.

Corrective action: Remove the call from the application.

On MVS/ESA, for a CICS or IMS application, issue the appropriate CICS or IMS call instead.

MQRC_EXPIRY_ERROR
(2013, X'7DD') Expiry time not valid.

On an MQPUT or MQPUT1 call, the value specified for the *Expiry* field in the message descriptor MQMD is not valid.

Corrective action: Specify a value which is greater than zero, or the special value MQEI_UNLIMITED.

MQRC_FEEDBACK_ERROR
(2014, X'7DE') Feedback code not valid.

On an MQPUT or MQPUT1 call, the value specified for the *Feedback* field in the message descriptor MQMD is not valid. The value is outside both the range defined for system feedback codes and that defined for application feedback codes.

Corrective action: Specify a value in the range MQFB_SYSTEM_FIRST through MQFB_SYSTEM_LAST, or MQFB_APPL_FIRST through MQFB_APPL_LAST.

MQRC_FORMAT_ERROR
(2110, X'83E') Message format not valid.

On an MQGET call with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, one or both of the *CodedCharSetId* and *Encoding* fields in the message differs from the corresponding field in the *MsgDesc* parameter, but the message cannot be converted successfully due to an error associated with the message format. Possible errors include:

- A user-written exit with the name specified by the *Format* field in the message cannot be found.
- The format name in the message is MQFMT_NONE.
- The message contains data that is not consistent with the format definition.

The message is returned unconverted to the application issuing the MQGET call, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

Corrective action: Check the format name that was specified when the message was put. If this is not one of the built-in formats, check that a suitable exit with the same name as the format is available for the queue manager to load. Verify that the data in the message corresponds to the format expected by the exit.

MQRC_FUNCTION_ERROR
(2281, X'8E9') Function identifier not valid for service.

The function identifier *Function* specified on the MQZEP call is not valid for the service being configured.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Specify an MQZID_* value that is valid for the service being configured. Refer to the description of installable services in the *MQSeries Programmable System Management* book to determine which values are valid.

MQRC_GET_INHIBITED
(2016, X'7E0') Gets inhibited for the queue.

MQGET calls are currently inhibited for the queue (see the *InhibitGet* queue attribute described in “Attributes for all queues” on page 343), or for the queue to which this queue resolves (see “Attributes for alias queues” on page 365).

Corrective action: If the system design allows get requests to be inhibited for short periods, retry the operation later.

MQRC_GMO_ERROR
(2186, X'88A') 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 pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- The queue manager cannot copy the changed structure to application storage, even though the call is successful. This can occur, for example, if the pointer points to read-only storage.

Corrective action: Correct the definition of the MQGMO structure. Ensure that required input fields are correctly set.

MQRC_GROUP_ID_ERROR
(2258, X'8D2') Group identifier not valid.

An MQPUT or MQPUT1 call was issued to put a distribution-list message that is also a message in a group, a message segment, or has segmentation allowed, but an invalid combination of options and values was specified. All of the following are true:

- MQPMO_LOGICAL_ORDER is not specified in the *Options* field in MQPMO.
- Either there are too few MQPMR records provided by MQPMO, or the *GroupId* field is not present in the MQPMR records.
- One or more of the following flags is specified in the *MsgFlags* field in MQMD or MQMDE:

  MQMF_SEGMENTATION_ALLOWED
  MQMF_*_MSG_IN_GROUP
  MQMF_*_SEGMENT

- The *GroupId* field in MQMD or MQMDE is not MQGI_NONE.

This combination of options and values would result in the same group identifier being used for all of the destinations in the distribution list; this is not permitted by the queue manager.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify MQGI_NONE for the *GroupId* field in MQMD or MQMDE. Alternatively, if the call is MQPUT specify MQPMO_LOGICAL_ORDER in the *Options* field in MQPMO.

MQRC_HANDLE_NOT_AVAILABLE
(2017, X'7E1') No more handles available.

An MQOPEN or MQPUT1 call was issued, but the maximum number of open handles allowed for the current task has already been reached. Be aware that when a distribution list is specified on the MQOPEN or MQPUT1 call, each queue in the distribution list uses one handle.

On MVS/ESA, “task” means a CICS task, an MVS task, or an IMS-dependent region.

Corrective action: Check whether the application is issuing MQOPEN calls without corresponding MQCLOSE calls. If it is, modify the application to issue the MQCLOSE call for each open object as soon as that object is no longer needed.

Also check whether the application is specifying a distribution list containing a large number of queues that are consuming all of the available handles. If it is, increase the maximum number of handles that the task can use, or reduce the size of the distribution list. The maximum number of open handles that a task can use is given by the *MaxHandles* queue manager attribute (see “Attributes for the queue manager” on page 370).

MQRC_HCONFIG_ERROR
(2280, X'8E8') Configuration handle not valid.

The configuration handle *Hconfig* specified on the MQZEP call is not valid.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Specify the configuration handle that was provided to the service configuration function on the component initialization call. See the *MQSeries Programmable System Management* book for details of this call.

MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

The connection handle *Hconn* is not valid. This reason also occurs if the parameter pointer is not valid, or (for the MQCONN call) points to read-only storage. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Ensure that a successful MQCONN call is performed for the queue manager, and that an MQDISC call has not already been performed for it. Ensure that the handle is being used within its valid scope (see the MQCONN call described in "MQCONN – Connect queue manager" on page 261).

On MVS/ESA, also check that the application has been linked with the correct stub; this is CSQCSTUB for CICS applications, CSQBSTUB for batch applications, and CSQQSTUB for IMS applications. Also, the stub used must not belong to a release of MQSeries which is more recent than the release on which the application will run.

MQRC_HEADER_ERROR
(2142, X'85E') MQ header structure not valid.

The MQPUT or MQPUT1 call was used to put a message containing an MQ header structure, but the header structure is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQ header structure. Ensure that the fields are set correctly.

MQRC_HOBJ_ERROR
(2019, X'7E3') Object handle not valid.

The object handle *Hobj* is not valid. This reason also occurs if the parameter pointer is not valid, or (for the MQOPEN call) points to read-only storage. (It is not always possible to detect an invalid parameter pointer; if 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. Ensure that the handle is being used within its valid scope (see the MQOPEN call described in "MQOPEN – Open object" on page 297).

MQRC_IIH_ERROR
(2148, X'864') IMS information header structure not valid.

On an MQPUT or MQPUT1 call, the IMS information header structure MQIIH in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQIIH structure. Ensure that the fields are set correctly.

MQRC_INCOMPLETE_GROUP
(2241, X'8C1') Message group not complete.

An operation was attempted on a queue using a queue handle that had an incomplete message group. This reason code can arise in the following situations:

- On the MQPUT call, when the application attempts to put a message which is not in a group and specifies MQPMO_LOGICAL_ORDER. The call fails in this case.
- On the MQPUT call, when the application attempts to put a message which is not the next one in the group, does *not* specify MQPMO_LOGICAL_ORDER, but the previous MQPUT call for the queue handle did specify MQPMO_LOGICAL_ORDER. The call succeeds with completion code MQCC_WARNING in this case.
- On the MQGET call, when the application attempts to get a message which is not the next one in the group, does *not* specify MQGMO_LOGICAL_ORDER, but the previous MQGET call for the queue handle did specify MQGMO_LOGICAL_ORDER. The call succeeds with completion code MQCC_WARNING in this case.
- On the MQCLOSE call, when the application attempts to close the queue that has the incomplete message group. The call succeeds with completion code MQCC_WARNING.

If there is an incomplete logical message as well as an incomplete message group, reason code MQRC_INCOMPLETE_MSG is returned in preference to MQRC_INCOMPLETE_GROUP.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: If this reason code is expected, no corrective action is required. Otherwise, ensure that the MQPUT call for the last message in the group specifies MQMF_LAST_MSG_IN_GROUP.

MQRC_INCOMPLETE_MSG
(2242, X'8C2') Logical message not complete.

An operation was attempted on a queue using a queue handle that had an incomplete logical message. This reason code can arise in the following situations:

- On the MQPUT call, when the application attempts to put a message which is not a segment and specifies MQPMO_LOGICAL_ORDER. The call fails in this case.
- On the MQPUT call, when the application attempts to put a message which is not the next segment, does *not* specify MQPMO_LOGICAL_ORDER, but the previous MQPUT call for the queue handle did specify MQPMO_LOGICAL_ORDER. The call succeeds with completion code MQCC_WARNING in this case.
- On the MQGET call, when the application attempts to get a message which is not the next segment, does *not* specify MQGMO_LOGICAL_ORDER, but the previous MQGET call for the queue handle did specify MQGMO_LOGICAL_ORDER. The call succeeds with completion code MQCC_WARNING in this case.

- On the MQCLOSE call, when the application attempts to close the queue that has the incomplete logical message. The call succeeds with completion code MQCC_WARNING.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: If this reason code is expected, no corrective action is required. Otherwise, ensure that the MQPUT call for the last segment specifies MQMF_LAST_SEGMENT.

MQRC_INCONSISTENT_BROWSE
(2259, X'8D3') Inconsistent browse specification.

An MQGET call was issued with the MQGMO_BROWSE_NEXT option specified, but the specification of the MQGMO_LOGICAL_ORDER option for the call is different from the specification of that option for the previous call for the queue handle. Either both calls must specify MQGMO_LOGICAL_ORDER, or neither call must specify MQGMO_LOGICAL_ORDER.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Add or remove the MQGMO_LOGICAL_ORDER option as appropriate. Alternatively, to switch between logical order and physical order, specify the MQGMO_BROWSE_FIRST option to restart the scan from the beginning of the queue, and either omit or specify MQGMO_LOGICAL_ORDER as desired.

MQRC_INCONSISTENT_CCSIDS
(2243, X'8C3') Message segments have differing CCSIDs.

An MQGET call was issued specifying the MQGMO_COMPLETE_MSG option, but the message to be retrieved consists of two or more segments which have differing values for the *CodedCharSetId* field in MQMD. This can arise when the segments take different paths through the network, and some of those paths have MCA sender conversion enabled. The call succeeds with a completion code of MQCC_WARNING, but only the first few segments that have identical character-set identifiers are returned.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Remove the MQGMO_COMPLETE_MSG option from the MQGET call and retrieve the remaining message segments one by one.

MQRC_INCONSISTENT_ENCODINGS
(2244, X'8C4') Message segments have differing encodings.

An MQGET call was issued specifying the MQGMO_COMPLETE_MSG option, but the message to be retrieved consists of two or more segments which have differing values for the *Encoding* field in MQMD. This can arise when the segments take different paths through the network, and some of those paths have MCA sender conversion enabled. The call succeeds with a completion code of MQCC_WARNING, but only the first few segments that have identical encodings are returned.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Remove the MQGMO_COMPLETE_MSG option from the MQGET call and retrieve the remaining message segments one by one.

MQRC_INCONSISTENT_PERSISTENCE
(2185, X'889') Inconsistent persistence specification.

The MQPUT call was issued to put a message that has a value for the *Persistence* field in MQMD that is different from the previous message put using that queue handle. This is not permitted when the MQPMO_LOGICAL_ORDER option is specified and there is already a current message group or logical message. All messages in a group and all segments in a logical message must be persistent, or all must be nonpersistent.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Modify the application to ensure that all of the messages in the group or logical message are put with the same value for the *Persistence* field in MQMD.

MQRC_INCONSISTENT_UOW
(2245, X'8C5') Inconsistent unit-of-work specification.

One of the following applies:

- An MQPUT call was issued to put a message in a group or a segment of a logical message, but the value specified or defaulted for the MQPMO_SYNCPOINT option is not consistent with the current group and segment information retained by the queue manager for the queue handle.

  If the current call specifies MQPMO_LOGICAL_ORDER, the call fails. If the current call does not specify MQPMO_LOGICAL_ORDER, but the previous MQPUT call for the queue handle did, the call succeeds with completion code MQCC_WARNING.

- An MQGET call was issued to remove from the queue a message in a group or a segment of a logical message, but the value specified or defaulted for the MQGMO_SYNCPOINT option is not consistent with the current group and segment information retained by the queue manager for the queue handle.

  If the current call specifies MQGMO_LOGICAL_ORDER, the call fails. If the current call does not specify MQGMO_LOGICAL_ORDER, but the previous MQGET call for the queue handle did, the call succeeds with completion code MQCC_WARNING.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Modify the application to ensure that the same unit-of-work specification is used for all messages in the group, or all segments of the logical message.

MQRC_INHIBIT_VALUE_ERROR
(2020, X'7E4') Value for inhibit-get or inhibit-put queue attribute not valid.

On an MQSET call, the value specified for either the MQIA_INHIBIT_GET attribute or the MQIA_INHIBIT_PUT attribute is not valid.

Corrective action: Specify a valid value. See the *InhibitGet* or *InhibitPut* attribute described in “Attributes for all queues” on page 343.

MQRC_INITIALIZATION_FAILED
(2286, X'8EE') Initialization failed for an undefined reason.

This reason should be returned by an installable service component when the component is unable to complete initialization successfully.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Correct the error and retry the operation.

MQRC_INT_ATTR_COUNT_ERROR
(2021, X'7E5') Count of integer attributes not valid.

On an MQINQ or MQSET call, the *IntAttrCount* parameter is negative (MQINQ or MQSET), or smaller than the number of integer attribute selectors (MQIA_★) specified in the *Selectors* parameter (MQSET only). This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a value large enough for all selected integer attributes.

MQRC_INT_ATTR_COUNT_TOO_SMALL
(2022, X'7E6') Not enough space allowed for integer attributes.

On an MQINQ call, the *IntAttrCount* parameter is smaller than the number of integer attribute selectors (MQIA_★) specified in the *Selectors* parameter.

The call completes with MQCC_WARNING, with the *IntAttrs* array filled in 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.

MQRC_INT_ATTRS_ARRAY_ERROR
(2023, X'7E7') Integer attributes array not valid.

On an MQINQ or MQSET call, the *IntAttrs* parameter is not valid. The parameter pointer is not valid (MQINQ and MQSET), or points to read-only storage or to storage that is not as long as indicated by the *IntAttrCount* parameter (MQINQ only). (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

MQRC_INVALID_MSG_UNDER_CURSOR
(2246, X'8C6') Message under cursor not valid for retrieval.

An MQGET call was issued specifying the MQGMO_COMPLETE_MSG option with either MQGMO_MSG_UNDER_CURSOR or MQGMO_BROWSE_MSG_UNDER_CURSOR, but the message that is under the cursor has an MQMD with an *Offset* field that is greater than zero. Because MQGMO_COMPLETE_MSG was specified, the message is not valid for retrieval.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Reposition the browse cursor so that it is located on a message whose *Offset* field in MQMD is zero. Alternatively, remove the MQGMO_COMPLETE_MSG option.

MQRC_MATCH_OPTIONS_ERROR
(2247, X'8C7') Match options not valid.

An MQGET call was issued, but the value of the *MatchOptions* field in the *GetMsgOpts* parameter is not valid. Either an undefined option is specified, or a defined option which is not valid in the current circumstances is specified. In the latter case, it means that all of the following are true:

- MQGMO_LOGICAL_ORDER is specified.
- There is a current message group or logical message for the queue handle.
- Neither of the following options is specified:

  MQGMO_BROWSE_MSG_UNDER_CURSOR
  MQGMO_MSG_UNDER_CURSOR
- One or more of the MQMO_★ options is specified.
- The values of the fields in the *MsgDesc* parameter corresponding to the MQMO_★ options specified, differ from the values of those fields in the MQMD for the message to be returned next.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Ensure that only valid options are specified for the field.

MQRC_MAX_CONNS_LIMIT_REACHED
(2025, X'7E9') Maximum number of connections reached.

The MQCONN call was rejected because the maximum number of concurrent connections has been exceeded.

- On MVS/ESA, connection limits are applicable only to TSO and batch requests. The limits are determined by the customer using the following parameters of the CSQ6SYSP macro:
  - For TSO, IDFORE
  - For batch, IDBACK

  For more information, see the *MQSeries for MVS/ESA System Management Guide*.
- On OpenVMS, OS/2, Tandem NSK, UNIX systems, and Windows NT, this reason code can also occur on the MQOPEN call.
- On OS/400, this reason code does not occur.

Corrective Action: Either increase the size of the appropriate install parameter value, or reduce the number of concurrent connections.

MQRC_MD_ERROR
(2026, X'7EA') Message descriptor not valid.

MQMD 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 pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- The queue manager cannot copy the changed structure to application storage, even though the call is successful. This can occur, for example, if the pointer points to read-only storage.

Corrective action: Correct the definition of the message descriptor. Ensure that required input fields are correctly set.

MQRC_MDE_ERROR
(2248, X'8C8') Message descriptor extension not valid.

The MQMDE structure at the start of the application message data is not valid, for one of the following reasons:

- The *StrucId* mnemonic eye-catcher is not MQMDE_STRUC_ID.
- The *Version* field is less than MQMDE_VERSION_2.
- The *StrucLength* field is less than MQMDE_LENGTH_2, or (for *Version* equal to MQMDE_VERSION_2 only) greater than MQMDE_LENGTH_2.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the message descriptor extension. Ensure that required input fields are correctly set.

MQRC_MISSING_REPLY_TO_Q
(2027, X'7EB') Missing reply-to queue.

On an MQPUT or MQPUT1 call, the *ReplyToQ* field in the message descriptor MQMD is blank, but one or both of the following is true:

- A reply was requested (that is, MQMT_REQUEST was specified in the *MsgType* field of the message descriptor).
- A report message was requested in the *Report* field of the message descriptor.

Corrective action: Specify the name of the queue to which the reply message or report message is to be sent.

MQRC_MSG_FLAGS_ERROR
(2249, X'8C9') Message flags not valid.

An MQPUT or MQPUT1 call was issued, but the *MsgFlags* field in the message descriptor MQMD contains one or more message flags which are not recognized by the local queue manager. The message flags that cause this reason code to be returned depend on the destination of the message; see Appendix C, “Report options and message flags” on page 489 for more details.

This reason code can also occur in the *Feedback* field in the MQMD of a report message, or in the *Reason* field in the MQDLH structure of a message on the dead-letter queue; in both cases it indicates that the destination queue manager does not support one or more of the message flags specified by the sender of the message.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Do the following:

1. Ensure that the *MsgFlags* field in the message descriptor is initialized with a value when the message descriptor is declared, or is assigned a value prior to the MQPUT or MQPUT1 call.

   Specify MQMF_NONE if no message flags are needed.

2. Ensure that the message flags specified are ones which are documented in this book; see the *MsgFlags* field described in "MQMD – Message descriptor" on page 98 for valid message flags. Remove any message flags which are not documented in this book.

3. If multiple message flags are being set by adding the individual message flags together, ensure that the same message flag is not added twice.

MQRC_MSG_SEQ_NUMBER_ERROR
(2250, X'8CA') Message sequence number not valid.

An MQGET, MQPUT, or MQPUT1 call was issued, but the value of the *MsgSeqNumber* field in the MQMD or MQMDE structure is less than one or greater than 999 999 999.

This error can also occur on the MQPUT call if the *MsgSeqNumber* field would have become greater than 999 999 999 as a result of the call.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify a value in the range 1 through 999 999 999. Do not attempt to create a message group containing more than 999 999 999 messages.

MQRC_MSG_TOO_BIG_FOR_CHANNEL
(2218, X'8AA') Message length greater than maximum for channel.

A message was put to a remote queue, but the message is larger than the maximum message length allowed by the channel. This reason code is returned in the *Feedback* field in the message descriptor of a report message.

On MVS/ESA, this return code is issued only if you are not using CICS for distributed queuing. Otherwise, MQRC_MSG_TOO_BIG_FOR_Q_MGR is issued.

Corrective action: Check the channel definitions. Increase the maximum message length that the channel can accept, or break the message into several smaller messages.

MQRC_MSG_TOO_BIG_FOR_Q
(2030, X'7EE') Message length greater than maximum for queue.

An MQPUT or MQPUT1 call was issued to put a message on a queue, but the message was too long for the queue and MQMF_SEGMENTATION_ALLOWED was not specified in the *MsgFlags* field in MQMD. If segmentation is not allowed, the length of the message cannot exceed the lesser of the queue and queue-manager *MaxMsgLength* attributes.

This reason code can also occur when MQMF_SEGMENTATION_ALLOWED *is* specified, but the nature of the data present in the message prevents the queue manager splitting it into segments that are small enough to place on the queue:

- For a user-defined format, the smallest segment that the queue manager can create is 16 bytes.

- For a built-in format, the smallest segment that the queue manager can create depends on the particular format, but is greater than 16 bytes in all cases other than MQFMT_STRING (for MQFMT_STRING the minimum segment size is 16 bytes).

MQRC_MSG_TOO_BIG_FOR_Q can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

Corrective action: Check whether the *BufferLength* parameter is specified correctly; if it is, do one of the following:

- Increase the value of the queue's *MaxMsgLength* attribute; the queue-manager's *MaxMsgLength* attribute may also need increasing.
- Break the message into several smaller messages.
- Specify MQMF_SEGMENTATION_ALLOWED in the *MsgFlags* field in MQMD; this will allow the queue manager to break the message into segments.

MQRC_MSG_TOO_BIG_FOR_Q_MGR
(2031, X'7EF') Message length greater than maximum for queue manager.

An MQPUT or MQPUT1 call was issued to put a message on a queue, but the message was too long for the queue manager and MQMF_SEGMENTATION_ALLOWED was not specified in the *MsgFlags* field in MQMD. If segmentation is not allowed, the length of the message cannot exceed the lesser of the queue and queue-manager *MaxMsgLength* attributes.

This reason code can also occur when MQMF_SEGMENTATION_ALLOWED *is* specified, but the nature of the data present in the message prevents the queue manager splitting it into segments that are small enough for the queue-manager limit:

- For a user-defined format, the smallest segment that the queue manager can create is 16 bytes.
- For a built-in format, the smallest segment that the queue manager can create depends on the particular format, but is greater than 16 bytes in all cases other than MQFMT_STRING (for MQFMT_STRING the minimum segment size is 16 bytes).

MQRC_MSG_TOO_BIG_FOR_Q_MGR can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

This reason also occurs if a channel, through which the message is to pass, has restricted the maximum message length to a value that is actually less than that supported by the queue manager, and the message length is greater than this value.

On MVS/ESA, this return code is issued only if you are using CICS for distributed queuing. Otherwise, MQRC_MSG_TOO_BIG_FOR_CHANNEL is issued.

Corrective action: Check whether the *BufferLength* parameter is specified correctly; if it is, do one of the following:

- Increase the value of the queue-manager's *MaxMsgLength* attribute; the queue's *MaxMsgLength* attribute may also need increasing.
- Break the message into several smaller messages.
- Specify MQMF_SEGMENTATION_ALLOWED in the *MsgFlags* field in MQMD; this will allow the queue manager to break the message into segments.
- Check the channel definitions.

MQRC_MSG_TYPE_ERROR
(2029, X'7ED') Message type in message descriptor not valid.

On an MQPUT or MQPUT1 call, the value specified for the *MsgType* field in the message descriptor (MQMD) is not valid.

Corrective action: Specify a valid value. See the *MsgType* field described in "MQMD – Message descriptor" on page 98 for details.

MQRC_MULTIPLE_REASONS
(2136, X'858') Multiple reason codes returned.

An MQOPEN, MQPUT or MQPUT1 call was issued to open a distribution list or put a message to a distribution list, but the result of the call was not the same for all of the destinations in the list. One of the following applies:

- The call succeeded for some of the destinations but not others. The completion code is MQCC_WARNING in this case.
- The call failed for all of the destinations, but for differing reasons. The completion code is MQCC_FAILED in this case.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Examine the MQRR response records to identify the destinations for which the call failed, and the reason for the failure. Ensure that sufficient response records are provided by the application on the call to enable the error(s) to be determined. For the MQPUT1 call, the response records must be specified using the MQOD structure, and not the MQPMO structure.

MQRC_NAME_IN_USE
(2201, X'899') Name in use.

An MQOPEN call was issued to create a dynamic queue, but a queue with the same name as the dynamic queue already exists. The existing queue is one that is logically deleted, but for which there are still one or more open handles. For more information, see "MQCLOSE – Close object" on page 248.

This reason code occurs only on MVS/ESA.

Corrective action: Either ensure that all handles for the previous dynamic queue are closed, or ensure that the name of the new queue is unique; see the description for reason code MQRC_OBJECT_ALREADY_EXISTS.

MQRC_NAME_NOT_VALID_FOR_TYPE
(2194, X'892') Object name not valid for object type.

An MQOPEN call was issued to open the queue manager definition, but the *ObjectName* field in the *ObjDesc* parameter is not blank.

Corrective action: Ensure that the *ObjectName* field is set to blanks.

MQRC_NO_EXTERNAL_PARTICIPANTS
(2121, X'849') No participating resource managers registered.

An MQBEGIN call was issued to start a unit of work coordinated by the queue manager, but no participating resource managers have been registered with the queue manager. As a result, only changes to MQ resources can be coordinated by the queue manager in the unit of work.

This reason code occurs in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Corrective action: If the application does not require non-MQ resources to participate in the unit of work, this reason code can be ignored or the MQBEGIN call removed. Otherwise consult your system support programmer to determine why the required resource managers have not been registered with the queue manager; the queue manager's configuration file may be in error.

MQRC_NO_MSG_AVAILABLE
(2033, X'7F1') No message available.

An MQGET call was issued, but there is no message on the queue satisfying the selection criteria specified in MQMD (the *MsgId* and *CorrelId* fields), and in MQGMO (the *Options* and *MatchOptions* fields). Either the MQGMO_WAIT option was not specified, or the time interval specified by the *WaitInterval* field in MQGMO has expired. This reason is also returned for an MQGET call for browse, when the end of the queue has been reached.

Corrective action: If this is an expected condition, no corrective action is required.

If this is an unexpected condition, check whether the message was put on the queue successfully, and whether the options controlling the selection criteria are specified correctly. All of the following can affect the eligibility of a message for return on the MQGET call:

MQGMO_LOGICAL_ORDER
MQGMO_ALL_MSGS_AVAILABLE
MQGMO_ALL_SEGMENTS_AVAILABLE
MQGMO_COMPLETE_MSG
MQMO_MATCH_MSG_ID
MQMO_MATCH_CORREL_ID
MQMO_MATCH_GROUP_ID
MQMO_MATCH_MSG_SEQ_NUMBER
MQMO_MATCH_OFFSET
MsgId field
CorrelId field

Consider waiting longer for the message.

MQRC_NO_MSG_LOCKED
(2209, X'8A1') No message locked.

An MQGET call was issued with the MQGMO_UNLOCK option, but no message was currently locked.

On MVS/ESA, this reason code does not occur.

Corrective action: Check that a message was locked by an earlier MQGET call with the MQGMO_LOCK option for the same handle, and that no intervening call has caused the message to become unlocked.

MQRC_NO_MSG_UNDER_CURSOR
(2034, X'7F2') Browse cursor not positioned on message.

An MQGET call was issued with either the MQGMO_MSG_UNDER_CURSOR or the MQGMO_BROWSE_MSG_UNDER_CURSOR option. However, the browse cursor is not positioned at a retrievable message. This is caused by one of the following:

- The cursor is positioned logically before the first message (as it is before the first MQGET call with a browse option has been successfully performed), or
- The message the browse cursor was positioned on has been locked or removed from the queue (probably by some other application) since the browse operation was performed.
- The message the browse cursor was positioned on has expired.

Corrective action: Check the application logic. This may be an expected reason if the application design allows multiple servers to compete for messages after browsing. Consider also using the MQGMO_LOCK option with the preceding browse MQGET call.

MQRC_NONE
(0, X'000') No reason to report.

The call completed normally. The completion code (*CompCode*) is MQCC_OK.

Corrective action: None.

MQRC_NOT_AUTHORIZED
(2035, X'7F3') Not authorized for access.

The user is not authorized to perform the operation attempted:

- On an MQCONN call, the user is not authorized to connect to the queue manager.

  On MVS/ESA, for CICS applications, MQRC_CONNECTION_NOT_AUTHORIZED is issued instead.

- On an MQOPEN or MQPUT1 call, the user is not authorized to open the object for the option(s) specified.

  On MVS/ESA, if the object being opened is a model queue, this reason also arises if the user is not authorized to create a dynamic queue with the required name.

- On an MQCLOSE call, the user is not authorized to delete the object, which is a permanent dynamic queue, and the *Hobj* parameter specified on the MQCLOSE call is not the handle returned by the MQOPEN call that created the queue.

This reason code can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

Corrective action: Ensure that the correct queue manager or object was specified, and that appropriate authority exists.

On MVS/ESA, to determine for which object you are not authorized, you can use the violation messages issued by the External Security Manager.

MQRC_NOT_CONVERTED
(2119, X'847') Application message data not converted.

On an MQGET call with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, an error occurred during conversion of the data in the message. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

This error may also indicate that a parameter to the data-conversion service is not supported.

Corrective action: Check that the message data is correctly described by the *Format*, *CodedCharSetId* and *Encoding* parameters that were specified when the message was put. Also check that these values, and the *CodedCharSetId* and *Encoding* specified in the *MsgDesc* parameter on the MQGET call, are supported for queue-manager conversion. If the required conversion is not supported, conversion must be carried out by the application.

MQRC_NOT_OPEN_FOR_BROWSE
(2036, X'7F4') Queue not open for browse.

An MQGET call was issued with one of the following options:

MQGMO_BROWSE_FIRST
MQGMO_BROWSE_NEXT
MQGMO_BROWSE_MSG_UNDER_CURSOR
MQGMO_MSG_UNDER_CURSOR

but the queue had not been opened for browse.

Corrective action: Specify MQOO_BROWSE when the queue is opened.

MQRC_NOT_OPEN_FOR_INPUT
(2037, X'7F5') Queue not open for input.

An MQGET call was issued to retrieve a message from a queue, but the queue had not been opened for input.

Corrective action: Specify one of the following when the queue is opened:

MQOO_INPUT_SHARED
MQOO_INPUT_EXCLUSIVE
MQOO_INPUT_AS_Q_DEF

MQRC_NOT_OPEN_FOR_INQUIRE
(2038, X'7F6') Queue not open for inquire.

An MQINQ call was issued to inquire object attributes, but the object had not been opened for inquire.

Corrective action: Specify MQOO_INQUIRE when the object is opened.

MQRC_NOT_OPEN_FOR_OUTPUT
(2039, X'7F7') Queue not open for output.

An MQPUT call was issued to put a message on a queue, but the queue had not been opened for output.

Corrective action: Specify MQOO_OUTPUT when the queue is opened.

MQRC_NOT_OPEN_FOR_PASS_ALL
(2093, X'82D') Queue not open for pass all context.

An MQPUT call was issued with the MQPMO_PASS_ALL_CONTEXT option specified in the *PutMsgOpts* parameter, but the queue had not been opened with the MQOO_PASS_ALL_CONTEXT option.

Corrective action: Specify MQOO_PASS_ALL_CONTEXT (or another option that implies it) when the queue is opened.

MQRC_NOT_OPEN_FOR_PASS_IDENT
(2094, X'82E') Queue not open for pass identity context.

An MQPUT call was issued with the MQPMO_PASS_IDENTITY_CONTEXT option specified in the *PutMsgOpts* parameter, but the queue had not been opened with the MQOO_PASS_IDENTITY_CONTEXT option.

Corrective action: Specify MQOO_PASS_IDENTITY_CONTEXT (or another option that implies it) when the queue is opened.

MQRC_NOT_OPEN_FOR_SET
(2040, X'7F8') Queue not open for set.

An MQSET call was issued to set queue attributes, but the queue had not been opened for set.

Corrective action: Specify MQOO_SET when the object is opened.

MQRC_NOT_OPEN_FOR_SET_ALL
(2095, X'82F') Queue not open for set all context.

An MQPUT call was issued with the MQPMO_SET_ALL_CONTEXT option specified in the *PutMsgOpts* parameter, but the queue had not been opened with the MQOO_SET_ALL_CONTEXT option.

Corrective action: Specify MQOO_SET_ALL_CONTEXT when the queue is opened.

MQRC_NOT_OPEN_FOR_SET_IDENT
(2096, X'830') Queue not open for set identity context.

An MQPUT call was issued with the MQPMO_SET_IDENTITY_CONTEXT option specified in the *PutMsgOpts* parameter, but the queue had not been opened with the MQOO_SET_IDENTITY_CONTEXT option.

Corrective action: Specify MQOO_SET_IDENTITY_CONTEXT (or another option that implies it) when the queue is opened.

MQRC_OBJECT_ALREADY_EXISTS
(2100, X'834') Object already exists.

An MQOPEN call was issued to create a dynamic queue, but a queue with the same name as the dynamic queue already exists.

On MVS/ESA, a rare “race condition” can also give rise to this reason code; see the description of reason code MQRC_NAME_IN_USE for more details.

Corrective action: If supplying a dynamic queue name in full, ensure that it obeys the naming conventions for dynamic queues; if it does, either supply a different name, or delete the existing queue if it is no longer required. Alternatively, allow the queue manager to generate the name.

If the queue manager is generating the name (either in part or in full), reissue the MQOPEN call.

MQRC_OBJECT_CHANGED
(2041, X'7F9') Object definition changed since opened.

Since the *Hobj* handle used on this call was returned by the MQOPEN call, object definitions that affect this object have been changed. See “MQOPEN – Open object” on page 297 for more information.

This reason does not occur if the object handle is specified in the *Context* field of the *PutMsgOpts* parameter on the MQPUT or MQPUT1 call.

Corrective action: Issue an MQCLOSE call to return the handle to the system. It is then usually sufficient to reopen the object and retry the operation. However, if the object definitions are critical to the application logic, an MQINQ call can be used after reopening the object, to find out what has changed.

MQRC_OBJECT_DAMAGED
(2101, X'835') Object damaged.

The object accessed by the call is damaged and cannot be used. For example, this may be because the definition of the object in main storage is not consistent, or because it differs from the definition of the object on disk, or because the definition on disk cannot be read.

The object cannot be used until the problem is corrected. The object can be deleted, although it may not be possible to delete the associated user space.

On MVS/ESA, this reason code does not occur.

Corrective action: It may be necessary to stop and restart the queue manager, or to restore the queue-manager data from back-up storage.

On OpenVMS, OS/2, OS/400, Tandem NSK, and UNIX systems, consult the FFST record to obtain more detail about the problem.

MQRC_OBJECT_IN_USE
(2042, X'7FA') Object already open with conflicting options.

An MQOPEN call was 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; it also arises if the request is for exclusive input, but the object is already open for input (of any sort).

Note: MCAs for receiver channels may keep the destination queues open even when messages are not being transmitted; this results in the queues appearing to be “in use.”

On MVS/ESA, this reason can also occur for an MQOPEN or MQPUT1 call, if the object to be opened (which can be a queue, or for MQOPEN a namelist or process object) is in the process of being deleted.

Corrective action: System design should specify whether an application is to wait and retry, or take other action.

MQRC_OBJECT_NAME_ERROR
(2152, X'868') Object name not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *RecsPresent* field in MQOD is greater than zero), but the *ObjectName* field is neither blank nor the null string.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: If it is intended to open a distribution list, set the *ObjectName* field to blanks or the null string. If it is not intended to open a distribution list, set the *RecsPresent* field to zero.

MQRC_OBJECT_Q_MGR_NAME_ERROR
(2153, X'869') Object queue-manager name not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *RecsPresent* field in MQOD is greater than zero), but the *ObjectQMgrName* field is neither blank nor the null string.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: If it is intended to open a distribution list, set the *ObjectQMgrName* field to blanks or the null string. If it is not intended to open a distribution list, set the *RecsPresent* field to zero.

MQRC_OBJECT_RECORDS_ERROR
(2155, X'86B') Object records not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *RecsPresent* field in MQOD is greater than zero), but the MQOR object records are not specified correctly. One of the following applies:

- *ObjectRecOffset* is zero and *ObjectRecPtr* is the null pointer or zero.
- *ObjectRecOffset* is not zero and *ObjectRecPtr* is neither the null pointer nor zero.
- *ObjectRecPtr* is not a valid pointer.
- *ObjectRecPtr* or *ObjectRecOffset* points to storage that is not accessible.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Ensure that one of *ObjectRecOffset* and *ObjectRecPtr* is zero and the other nonzero. Ensure that the field used points to accessible storage.

MQRC_OBJECT_TYPE_ERROR
(2043, X'7FB') Object type not valid.

On the MQOPEN or MQPUT1 call, the *ObjectType* field in the object descriptor MQOD specifies a value which is not valid. For the MQPUT1 call, the object type must be MQOT_Q.

Corrective action: Specify a valid object type.

MQRC_OD_ERROR
(2044, X'7FC') Object descriptor structure not valid.

On the MQOPEN or MQPUT1 call, the object descriptor MQOD 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. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- The queue manager cannot copy the changed structure to application storage, even though the call is successful. This can occur, for example, if the pointer points to read-only storage.

Corrective action: Correct the definition of the object descriptor. Ensure that required input fields are set correctly.

MQRC_OFFSET_ERROR
(2251, X'8CB') Message segment offset not valid.

An MQPUT or MQPUT1 call was issued, but the value of the *Offset* field in the MQMD or MQMDE structure is less than zero or greater than 999 999 999.

This error can also occur on the MQPUT call if the *Offset* field would have become greater than 999 999 999 as a result of the call.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify a value in the range 0 through 999 999 999. Do not attempt to create a message segment which would extend beyond an offset of 999 999 999.

MQRC_OPEN_FAILED
(2137, X'859') Queue not opened successfully.

An MQPUT call was issued to put a message to a distribution list, but the message could not be sent to the destination to which this reason code applies because that destination was not opened successfully by the MQOPEN call. This reason occurs only in the *Reason* field of the MQRR response record.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Examine the MQRR response records specified on the MQOPEN call to determine the reason that the queue failed to open. Ensure that sufficient response records are provided by the application on the call to enable the error(s) to be determined.

MQRC_OPTION_NOT_VALID_FOR_TYPE
(2045, X'7FD') Option not valid for object type.

On an MQOPEN or MQCLOSE call, an option is specified that is not valid for the type of object or queue being opened or closed. For the MQOPEN call, this includes both an option that is inappropriate to the object type (for example, MQOO_OUTPUT for an MQOT_PROCESS object), and one that is unsupported for the queue type (for example, MQOO_INQUIRE for a remote queue that has no local definition).

This reason also occurs on the MQOPEN call if one of the following is true:

- the queue name is resolved through a cell directory, or
- *ObjectQMgrName* in the object descriptor specifies the name of a local definition of a remote queue (in order to specify a queue-manager alias), and the queue named in the *RemoteQMgrName* attribute of the definition is the name of the local queue manager,

and the options include one of the following:

MQOO_INPUT_AS_Q_DEF
MQOO_INPUT_SHARED
MQOO_INPUT_EXCLUSIVE
MQOO_BROWSE
MQOO_INQUIRE
MQOO_SET

For the MQCLOSE call, this reason code occurs when the MQCO_DELETE or MQCO_DELETE_PURGE option was specified, but the queue is not a dynamic queue.

Corrective action: Specify the correct option; see Table 60 on page 302 for open options, and Table 59 on page 250 for close options. For the MQCLOSE call, either correct the option or change the definition type of the model queue that was used to create the queue.

MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.

The *Options* parameter or field contains options which are not valid, or a combination of options which is not valid.

- For the MQOPEN, MQCLOSE, and MQXCNVC calls, *Options* is a separate parameter on the call.

  This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

- For the MQBEGIN, MQCONNX, MQGET, MQPUT, and MQPUT1 calls, *Options* is a field in the relevant options structure (MQBO, MQCNO, MQGMO, or MQPMO).

Corrective action: Specify valid options. Check the description of the *Options* parameter or field to determine which options and combinations of options are valid. If multiple options are being set by adding the individual options together, ensure that the same option is not added twice.

MQRC_ORIGINAL_LENGTH_ERROR
(2252, X'8CC') Original length not valid.

An MQPUT or MQPUT1 call was issued to put a report message which is reporting on a segment, but the *OriginalLength* field in the MQMD or MQMDE structure is either:

- Less than one (for a segment which is not the last segment), or
- Less than zero (for a segment which is the last segment)

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify a value which is greater than zero. Zero is valid only for the last segment.

MQRC_OUTCOME_MIXED
(2123, X'84B') Result of commit or back-out operation is mixed.

The queue manager is acting as the unit-of-work coordinator for a unit of work that involves other resource managers, but one of the following occurred:

- An MQCMIT or MQDISC call was issued to commit the unit of work, but one or more of the participating resource managers backed-out the unit of work instead of committing it. As a result, the outcome of the unit of work is mixed.
- An MQBACK call was issued to back out a unit of work, but one or more of the participating resource managers had already committed the unit of work.

This reason code occurs in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Corrective action: Examine the queue-manager error logs for messages relating to the mixed outcome; these messages identify the resource managers that are affected. Use procedures local to the affected resource managers to resynchronize the resources.

Note: This reason code does not prevent the application initiating further units of work.

MQRC_OUTCOME_PENDING
(2124, X'84C') Result of commit operation is pending.

The queue manager is acting as the unit-of-work coordinator for a unit of work that involves other resource managers, and an MQCMIT or MQDISC call was issued to commit the unit of work, but one or more of the participating resource managers has not confirmed that the unit of work was committed successfully.

The completion of the commit operation will happen at some point in the future, but there remains the possibility that the outcome will be mixed.

This reason code occurs in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Corrective action: Use the normal error-reporting mechanisms to determine whether the outcome was mixed. If it was, take appropriate action to resynchronize the resources.

Note: This reason code does not prevent the application initiating further units of work.

MQRC_PAGESET_ERROR
(2193, X'891') Error accessing page set data set.

An error was encountered with the page set while attempting to access it for a locally defined queue. This could be because the queue is on a page set that does not exist. A console message is issued that tells you the number of the page set in error. For example if the error occured in the TEST job, and your user ID is ABCDEFG, the message is:

```
CSQI041I CSQIALLC JOB TEST USER ABCDEFG RECEIVED COMPCODE
         MQRC_PAGESET_ERROR ON PAGE SET 27
```

If this reason code occurs while attempting to delete a dynamic queue with MQCLOSE, the dynamic queue has not been deleted.

This reason code occurs only on MVS/ESA.

Corrective action: Check that the storage class for the queue maps to a valid page set using the DISPLAY Q(xx) STGCLASS, DISPLAY STGCLASS(xx), and DISPLAY USAGE PSID commands. If you are unable to resolve the problem, notify the system programmer who should:

- Collect the following diagnostic information:
  - A description of the actions that led to the error
  - A listing of the application program being run at the time of the error
  - Details of the page sets defined for use by MQSeries
- Attempt to re-create the problem, and take a system dump immediately after the error occurs
- Contact your IBM Support Center

MQRC_PAGESET_FULL
(2192, X'890') Page set data set full.

On an MQOPEN, MQPUT or MQPUT1 call, a page set data set was found to be full while attempting to open or put a message on a locally defined queue.

This reason code occurs only on MVS/ESA.

Corrective action: Check which queues contain messages and look for any looping programs that might be unnecessarily filling up queues. Otherwise, request the system programmer to increase the size of the page set data sets.

MQRC_PARTICIPANT_NOT_AVAILABLE
(2122, X'84A') Participating resource manager not available.

An MQBEGIN call was issued to start a unit of work coordinated by the queue manager, but one or more of the participating resource managers that had been registered with the queue manager is not available. As a result, changes to those resources cannot be coordinated by the queue manager in the unit of work.

This reason code occurs in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Corrective action: If the application does not require non-MQ resources to participate in the unit of work, this reason code can be ignored. Otherwise consult your system support programmer to determine why the required resource managers are not available. The resource manager may have been halted temporarily, or there may be an error in the queue manager's configuration file.

MQRC_PCF_ERROR
(2149, X'865') PCF structures not valid.

An MQPUT or MQPUT1 call was issued to put a message containing PCF data, but the length of the message does not equal the sum of the lengths of the PCF structures present in the message. This can occur for messages with the following format names:

MQFMT_ADMIN
MQFMT_EVENT

MQFMT_PCF

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Ensure that the length of the message specified on the MQPUT or MQPUT1 call equals the sum of the lengths of the PCF structures contained within the message data.

MQRC_PERSISTENCE_ERROR
(2047, X'7FF') Persistence not valid.

On an MQPUT or MQPUT1 call, the value specified for the *Persistence* field in the message descriptor MQMD is not valid.

Corrective action: Specify one of the following values:

MQPER_PERSISTENT
MQPER_NOT_PERSISTENT
MQPER_PERSISTENCE_AS_Q_DEF

MQRC_PERSISTENT_NOT_ALLOWED
(2048, X'800') Message on a temporary dynamic queue cannot be persistent.

On an MQPUT or MQPUT1 call, the value specified for the *Persistence* field in the message descriptor MQMD specifies MQPER_PERSISTENT, but the queue on which the message is being placed is a temporary dynamic queue. Persistent messages cannot be put on temporary queues.

This reason code can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

Corrective action: Specify MQPER_NOT_PERSISTENT if the message is to be placed on a temporary dynamic queue. If persistence is required, use a permanent dynamic queue, or a predefined queue.

Be aware that server applications are recommended to send reply messages (message type MQMT_REPLY) with the same persistence as the original request message (message type MQMT_REQUEST). If the request message is persistent, the reply queue specified in the *ReplyToQ* field in the message descriptor MQMD cannot be a temporary dynamic queue; a permanent dynamic or predefined queue must be used as the reply queue in this situation.

MQRC_PMO_ERROR
(2173, X'87D') 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 pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- The queue manager cannot copy the changed structure to application storage, even though the call is successful. This can occur, for example, if the pointer points to read-only storage.

Corrective action: Correct the definition of the MQPMO structure. Ensure that required input fields are correctly set.

MQRC_PMO_RECORD_FLAGS_ERROR
(2158, X'86E') Put message record flags not valid.

An MQPUT or MQPUT1 call was issued to put a message, but the *PutMsgRecFields* field in the MQPMO structure is not valid, for one of the following reasons:

- The field contains flags which are not valid.
- The message is being put to a distribution list, and put message records have been provided (that is, *RecsPresent* is greater than zero, and one of *PutMsgRecOffset* or *PutMsgRecPtr* is nonzero), but *PutMsgRecFields* has the value MQPMRF_NONE.
- MQPMRF_ACCOUNTING_TOKEN is specified without either MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Ensure that *PutMsgRecFields* is set with the appropriate MQPMRF_★ flags to indicate which fields are present in the put message records. If MQPMRF_ACCOUNTING_TOKEN is specified, ensure that either MQPMO_SET_IDENTITY_CONTEXT or MQPMO_SET_ALL_CONTEXT is also specified. Alternatively, set both *PutMsgRecOffset* and *PutMsgRecPtr* to zero.

MQRC_PRIORITY_ERROR
(2050, X'802') Message priority not valid.

On an MQPUT or MQPUT1 call, the value of the *Priority* field in the message descriptor MQMD is not valid.

Corrective action: Specify a value which is zero or greater, or the special value MQPRI_PRIORITY_AS_Q_DEF.

On MVS/ESA, specify a value in the range 0 through *MaxPriority* (see “Attributes for the queue manager” on page 370), or the special value MQPRI_PRIORITY_AS_Q_DEF.

MQRC_PRIORITY_EXCEEDS_MAXIMUM
(2049, X'801') Message Priority exceeds maximum value supported.

On an MQPUT or MQPUT1 call, the value of the *Priority* field in the message descriptor MQMD exceeds the maximum priority supported by the local queue manager (see the *MaxPriority* queue-manager attribute described in “Attributes for the queue manager” on page 370). The message is accepted by the queue manager, but is placed on the queue at the queue manager’s maximum priority. The *Priority* field in the message descriptor retains the value specified by the application that put the message.

On MVS/ESA, this reason code does not occur.

Corrective action: None required, unless this reason code was not expected by the application that put the message.

MQRC_PUT_INHIBITED
(2051, X'803') Put calls inhibited for the queue.

MQPUT and MQPUT1 calls are currently inhibited for the queue (see the

InhibitPut queue attribute described in “Attributes for all queues” on page 343), or for the queue to which this queue resolves (see “Attributes for alias queues” on page 365).

This reason code can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

Corrective action: If the system design allows put requests to be inhibited for short periods, retry the operation later.

MQRC_PUT_MSG_RECORDS_ERROR
(2159, X'86F') Put message records not valid.

An MQPUT or MQPUT1 call was issued to put a message to a distribution list, but the MQPMR put message records are not specified correctly. One of the following applies:

- *PutMsgRecOffset* is not zero and *PutMsgRecPtr* is neither the null pointer nor zero.
- *PutMsgRecPtr* is not a valid pointer.
- *PutMsgRecPtr* or *PutMsgRecOffset* points to storage that is not accessible.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Ensure that at least one of *PutMsgRecOffset* and *PutMsgRecPtr* is zero. Ensure that the field used points to accessible storage.

MQRC_Q_ALREADY_EXISTS
(2290, X'8F2') Queue object already exists.

This reason should be returned by the MQZ_INSERT_NAME installable service component when the queue specified by the *QName* parameter is already defined to the name service.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: None. See the *MQSeries Programmable System Management* book for details of this installable service.

MQRC_Q_DELETED
(2052, X'804') Queue has been deleted.

An *Hobj* queue handle specified on a call refers to a dynamic queue that has been deleted since the queue was opened. (See “MQCLOSE – Close object” on page 248 for information about the deletion of dynamic queues.)

On MVS/ESA, this can also occur with the MQOPEN and MQPUT1 calls if a dynamic queue is being opened, but the queue is in a logically-deleted state. See MQCLOSE for more information about this.

Corrective action: Issue an MQCLOSE call to return the handle and associated resources to the system (the MQCLOSE call will succeed in this case). Check the design of the application that caused the error.

MQRC_Q_DEPTH_HIGH
(2224, X'8B0') Queue depth high limit reached or exceeded.

An MQPUT or MQPUT1 call has caused the queue depth to be incremented to or above the limit specified in the *QDepthHighLimit* attribute.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_Q_DEPTH_LOW
(2225, X'8B1') Queue depth low limit reached or exceeded.

An MQGET call has caused the queue depth to be decremented to or below the limit specified in the *QDepthLowLimit* attribute.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_Q_FULL
(2053, X'805') Queue already contains maximum number of messages.

On an MQPUT or MQPUT1 call, the call failed because the queue is full, that is, it already contains the maximum number of messages possible (see the *MaxQDepth* local-queue attribute described in "Attributes for local queues and model queues" on page 348).

This reason code can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

Corrective action: Retry the operation later. Consider increasing the maximum depth for this queue, or arranging for more instances of the application to service the queue.

MQRC_Q_MGR_ACTIVE
(2222, X'8AE') Queue manager created.

This condition is detected when a queue manager becomes active.

On MVS/ESA, this event is not generated for the first start of a queue manager, only on subsequent restarts.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_Q_MGR_NAME_ERROR
(2058, X'80A') Queue manager name not valid or not known.

On an MQCONN call, the value specified for the *QMgrName* parameter is not valid. This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

On MVS/ESA, for CICS applications, this reason can occur on any call if the original connect specified an incorrect or unrecognized name.

This reason also occurs if an application attempts to connect to a queue manager within a group (see the *QMgrName* parameter of MQCONN), and either:

- Queue-manager groups are not supported (they are only supported for MQ client applications), or
- There is no queue-manager group with the specified name.

Corrective action: Use an all-blank name if possible, or verify that the name used is valid.

MQRC_Q_MGR_NOT_ACTIVE
(2223, X'8AE') Queue manager unavailable.

This condition is detected when a queue manager is requested to stop or quiesce.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_Q_MGR_NOT_AVAILABLE
(2059, X'80B') Queue manager not available for connection.

On an MQCONN call, the queue manager identified by the *QMgrName* parameter is not available for connection at this time.

- On MVS/ESA, for CICS applications, this reason can occur on any call if the original connect specified a queue manager whose name was recognized, but which is not available.
- On OS/400, this reason can also be returned by the MQOPEN and MQPUT1 calls, when MQHC_DEF_HCONN is specified for the *Hconn* parameter.

If the connection is from an MQ client application, this reason code can occur if there is an error with the client-connection or the corresponding server-connection channel definitions.

On MVS/ESA, this reason code can also occur if the optional MVS/ESA client attachment feature has not been installed.

This reason also occurs if an application attempts to connect to a queue manager within a group (see the *QMgrName* parameter of MQCONN), when none of the queue managers in the group is available for connection at this time.

Corrective action: Ensure that the queue manager has been started. If the connection is from a client application, check the channel definitions.

MQRC_Q_MGR_QUIESCING
(2161, X'871') Queue manager quiescing.

The application attempted to connect to the queue manager, but the queue manager is in the quiescing state.

On OS/400, and on MVS/ESA for CICS, the application either issued the MQCONN call, or issued the MQOPEN call when no connection was established.

This reason code also occurs if the queue manager is in the quiescing state and an application issues one of the following calls:

- MQOPEN, with MQOO_FAIL_IF_QUIESCING included in the *Options* parameter
- MQGET, with MQGMO_FAIL_IF_QUIESCING included in the *Options* field of the *GetMsgOpts* parameter
- MQPUT or MQPUT1, with MQPMO_FAIL_IF_QUIESCING included in the *Options* field of the *PutMsgOpts* parameter

Corrective action: The application should tidy up and stop. If the MQOO_FAIL_IF_QUIESCING, MQPMO_FAIL_IF_QUIESCING, and MQGMO_FAIL_IF_QUIESCING options are not used, the application may

continue working in order to complete and commit the current unit of work; but it should not start another unit of work.

MQRC_Q_MGR_STOPPING
(2162, X'872') Queue manager shutting down.

A call has been issued when the queue manager is shutting down. If the call is an MQGET call with the MQGMO_WAIT option, the wait has been canceled. No more message-queuing calls can be issued.

On MVS/ESA, the MQRC_CONNECTION_BROKEN reason may be returned instead if, as a result of system scheduling factors, the queue manager shuts down before the call completes.

Corrective action: The application should tidy up and stop. Applications should ensure that any uncommitted updates are backed out; any unit of work that is coordinated by the queue manager is backed out automatically.

MQRC_Q_NOT_EMPTY
(2055, X'807') Queue contains one or more messages or uncommitted put or get requests.

An MQCLOSE call was issued for a permanent dynamic queue, with either:

- The MQCO_DELETE option specified, but there are messages still on the queue, or
- The MQCO_DELETE or MQCO_DELETE_PURGE option specified, but there are uncommitted get or put calls outstanding against the queue.

See the usage notes pertaining to dynamic queues for the MQCLOSE call for more information.

This reason code is also returned from a Programmable Command Format (PCF) command to clear or delete a queue, if the queue contains uncommitted messages (or committed messages in the case of delete queue without the purge option).

Corrective action: Check why there might be messages on the queue. Be aware that the *CurrentQDepth* local-queue attribute might be zero even though there are one or more messages on the queue; this can happen if the messages have been retrieved as part of a unit of work which has not yet been committed. If the messages can be discarded, try using the MQCLOSE call with the MQCO_DELETE_PURGE option. Consider retrying the call later.

MQRC_Q_SERVICE_INTERVAL_HIGH
(2226, X'8B2') Queue service interval high.

No successful gets or puts have been detected within an interval which is greater than the limit specified in the *QServiceInterval* attribute.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_Q_SERVICE_INTERVAL_OK
(2227, X'8B3') Queue service interval ok.

A successful get has been detected within an interval which is less than or equal to the limit specified in the *QServiceInterval* attribute.

Corrective action: None. This reason code is only used to identify the corresponding event message.

MQRC_Q_SPACE_NOT_AVAILABLE
(2056, X'808') No space available on disk for queue.

An MQPUT or MQPUT1 call was issued, but there is no space available for the queue on disk or other storage device.

This reason code can also occur in the *Feedback* field in the message descriptor of a report message; in this case it indicates that the error was encountered by a message channel agent when it attempted to put the message on a remote queue.

On MVS/ESA, this reason code does not occur.

Corrective action: Check whether an application is putting messages in an infinite loop. If not, make more disk space available for the queue.

On OS/400, the space available for a queue is limited to 320 MB. If this limit has been reached, consider redesigning the application to reduce the number or size of messages on a single queue, or start more server instances.

MQRC_Q_TYPE_ERROR
(2057, X'809') Queue type not valid.

One of the following occurred:

- On an MQOPEN call, the *ObjectQMgrName* field in the object descriptor MQOD or object record MQOR specifies the name of a local definition of a remote queue (in order to specify a queue-manager alias), and in that local definition the *RemoteQMgrName* attribute is the name of the local queue manager. However, the *ObjectName* field in MQOD or MQOR specifies the name of a model queue on the local queue manager; this is not allowed. See the *MQSeries Application Programming Guide* for more information.
- On an MQPUT1 call, the object descriptor MQOD or object record MQOR specifies the name of a model queue.
- On a previous MQPUT or MQPUT1 call, the *ReplyToQ* field in the message descriptor specified the name of a model queue, but a model queue cannot be specified as the destination for reply or report messages. Only the name of a predefined queue, or the name of the *dynamic* queue created from the model queue, can be specified as the destination. In this situation the reason code MQRC_Q_TYPE_ERROR is returned in the *Reason* field of the MQDLH structure when the reply message or report message is placed on the dead-letter queue.

Corrective action: Specify a valid queue.

MQRC_RECS_PRESENT_ERROR
(2154, X'86A') Number of records present not valid.

An MQOPEN or MQPUT1 call was issued, but the call failed for one of the following reasons:

- *RecsPresent* in MQOD is less than zero.
- *ObjectType* in MQOD is not MQOT_Q, and *RecsPresent* is not zero. *RecsPresent* must be zero if the object being opened is not a queue.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: If it is intended to open a distribution list, set the *ObjectType* field to MQOT_Q and *RecsPresent* to the number of destinations in the list. If it is not intended to open a distribution list, set the *RecsPresent* field to zero.

MQRC_REMOTE_Q_NAME_ERROR
(2184, X'888') Remote queue name not valid.

On an MQOPEN or MQPUT1 call, one of the following occurred:

- A local definition of a remote queue (or an alias to one) was specified, but the *RemoteQName* attribute in the remote queue definition is entirely blank. Note that this error occurs even if the *XmitQName* in the definition is not blank.
- The *ObjectQMgrName* field in the object descriptor was not blank and not the name of the local queue manager, but the *ObjectName* field is blank.

Corrective action: Alter the local definition of the remote queue and supply a valid remote queue name, or supply a nonblank *ObjectName* in the object descriptor, as appropriate.

MQRC_REPORT_OPTIONS_ERROR
(2061, X'80D') Report options in message descriptor not valid.

An MQPUT or MQPUT1 call was issued, but the *Report* field in the message descriptor MQMD contains one or more options which are not recognized by the local queue manager. The options that cause this reason code to be returned depend on the destination of the message; see Appendix C, “Report options and message flags” on page 489 for more details.

This reason code can also occur in the *Feedback* field in the MQMD of a report message, or in the *Reason* field in the MQDLH structure of a message on the dead-letter queue; in both cases it indicates that the destination queue manager does not support one or more of the report options specified by the sender of the message.

Corrective action: Do the following:

1. Ensure that the *Report* field in the message descriptor is initialized with a value when the message descriptor is declared, or is assigned a value prior to the MQPUT or MQPUT1 call.

   Specify MQRO_NONE if no report options are required.

2. Ensure that the report options specified are ones which are documented in this book; see the *Report* field described in “MQMD – Message descriptor” on page 98 for valid report options. Remove any report options which are not documented in this book.

3. If multiple report options are being set by adding the individual report options together, ensure that the same report option is not added twice.

4. Check that conflicting report options are not specified. For example, do not add both MQRO_EXCEPTION and MQRO_EXCEPTION_WITH_DATA to the *Report* field; only one of these can be specified.

MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.

There are insufficient system resources to complete the call successfully.

On MVS/ESA, this reason code does not occur.

Corrective action: Run the application when the machine is less heavily loaded.

On OpenVMS, OS/2, OS/400, Tandem NSK, and UNIX systems, consult the FFST record to obtain more detail about the problem.

MQRC_RESPONSE_RECORDS_ERROR
(2156, X'86C') Response records not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *RecsPresent* field in MQOD is greater than zero), but the MQRR response records are not specified correctly. One of the following applies:

- *ResponseRecOffset* is not zero and *ResponseRecPtr* is neither the null pointer nor zero.
- *ResponseRecPtr* is not a valid pointer.
- *ResponseRecPtr* or *ResponseRecOffset* points to storage that is not accessible.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Ensure that at least one of *ResponseRecOffset* and *ResponseRecPtr* is zero. Ensure that the field used points to accessible storage.

MQRC_RMH_ERROR
(2220, X'8AC') Reference message header structure not valid.

On an MQPUT or MQPUT1 call, the reference message header structure MQRMH in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQRMH structure. Ensure that the fields are set correctly.

MQRC_SECOND_MARK_NOT_ALLOWED
(2062, X'80E') A message is already marked.

The *Options* field in the MQGMO specifies MQGMO_MARK_SKIP_BACKOUT, but a message has already been marked within this unit of work. Only one marked message is allowed within one unit of work.

This reason code occurs only on MVS/ESA.

Corrective action: Ask for only one message to be marked.

MQRC_SECURITY_ERROR
(2063, X'80F') Security error occurred.

An MQCONN, MQOPEN, MQPUT1, or MQCLOSE call was issued, but it failed because a security error occurred.

- On MVS/ESA, the security error was returned by the External Security Manager.
- On OS/400, this reason code is not returned by the MQCONN call.

Corrective Action: Note the error from the security manager, and contact your system programmer or security administrator.

On OS/400, the FFST log will contain the error information.

MQRC_SEGMENT_LENGTH_ZERO
(2253, X'8CD') Length of data in message segment is zero.

An MQPUT or MQPUT1 call was issued to put the first or intermediate segment of a logical message, but the length of the application message data in the segment (excluding any MQ headers that may be present) is zero. The length must be at least one for the first or intermediate segment.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Check the application logic to ensure that segments are put with a length of one or greater. Only the last segment of a logical message is permitted to have a zero length.

MQRC_SELECTOR_COUNT_ERROR
(2065, X'811') Count of selectors not valid.

On an MQINQ or MQSET call, the *SelectorCount* parameter specifies a value that is not valid. This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a value in the range 0 through 256.

MQRC_SELECTOR_ERROR
(2067, X'813') Attribute selector not valid.

On an MQINQ or MQSET call, a selector in the *Selectors* array is either:

- not valid, or
- not applicable to the type of the object whose attributes are being inquired or set, or
- (MQSET only) not an attribute which can be set.

This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Ensure that the value specified for the selector is valid for the object type represented by *Hobj*. For the MQSET call, also ensure that the selector represents an integer attribute that can be set.

MQRC_SELECTOR_LIMIT_EXCEEDED
(2066, X'812') Count of selectors too big.

On an MQINQ or MQSET call, the *SelectorCount* parameter specifies a value that is larger than the maximum supported (256).

Corrective action: Reduce the number of selectors specified on the call; the valid range is 0 through 256.

MQRC_SELECTOR_NOT_FOR_TYPE
(2068, X'814') Selector not applicable to queue type.

On the MQINQ call, one or more selectors in the *Selectors* array is not applicable to the type of the queue whose attributes are being inquired. The call completes with MQCC_WARNING, with the attribute values for the inapplicable selectors set as follows:

- For integer attributes, the corresponding elements of *IntAttrs* are set to MQIAV_NOT_APPLICABLE.
- For character attributes, the appropriate parts of the *CharAttrs* string are set to a character string consisting entirely of asterisks (*).

Corrective action: Verify that the selector specified is the one that was intended.

MQRC_SERVICE_ERROR
(2289, X'8F1') Unexpected error occurred accessing service.

This reason should be returned by an installable service component when the component encounters an unexpected error.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Correct the error and retry the operation.

MQRC_SERVICE_NOT_AVAILABLE
(2285, X'8ED') Underlying service not available.

This reason should be returned by an installable service component when the requested action cannot be performed because the required underlying service is not available.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Make the underlying service available.

MQRC_SIGNAL_OUTSTANDING
(2069, X'815') Signal outstanding for this handle.

An MQGET call was issued with either the MQGMO_SET_SIGNAL or MQGMO_WAIT option, but there is already a signal outstanding for the queue handle *Hobj*.

This reason code occurs only in the following environments: MVS/ESA, 32-bit Windows.

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.

MQRC_SIGNAL_REQUEST_ACCEPTED
(2070, X'816') No message returned (but signal request accepted).

An MQGET call was issued specifying MQGMO_SET_SIGNAL in the *GetMsgOpts* parameter, but no suitable message was available; the call returns immediately. The application can now wait for the signal to be delivered.

- On MVS/ESA, the application should wait on the Event Control Block pointed to by the *Signal1* field.
- On 32-bit Windows, the application should wait for the signal Windows message to be delivered.

This reason code occurs only in the following environments: MVS/ESA, 32-bit Windows.

Corrective action: Wait for the signal; when it is delivered, check the signal to ensure that a message is now available. If it is, reissue the MQGET call.

On MVS/ESA, wait on the ECB pointed to by the *Signal1* field and, when it is posted, check it to ensure that a message is now available.

On 32-bit Windows, the application (thread) should continue executing its message loop.

MQRC_SIGNAL1_ERROR
(2099, X'833') Signal field not valid.

An MQGET call was issued, specifying MQGMO_SET_SIGNAL in the *GetMsgOpts* parameter, but the *Signal1* field is not valid.

- On MVS/ESA, the address contained in the *Signal1* field is not valid, or points to read-only storage. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)
- On 32-bit Windows, the window handle in the *Signal1* field is not valid.

This reason code occurs only in the following environments: MVS/ESA, 32-bit Windows.

Corrective action: Correct the setting of the *Signal1* field.

MQRC_SOURCE_BUFFER_ERROR
(2145, X'861') Source buffer parameter not valid.

On the MQXCNVC call, the *SourceBuffer* parameter pointer is not valid, or points to storage that cannot be accessed for the entire length specified by *SourceLength*. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a valid buffer.

MQRC_SOURCE_CCSID_ERROR
(2111, X'83F') Source coded character set identifier not valid.

The coded character-set identifier from which character data is to be converted is not valid or not supported.

This can occur on the MQGET call when the MQGMO_CONVERT option is included in the *GetMsgOpts* parameter; the coded character-set identifier in error is the *CodedCharSetId* field in the message being retrieved. In this case, the message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

This reason can also occur on the MQXCNVC call; the coded character-set identifier in error is the *SourceCCSID* parameter. Either the *SourceCCSID* parameter specifies a value which is not valid or not supported, or the *SourceCCSID* parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Check the character-set identifier that was specified when the message was put, or that was specified for the *SourceCCSID* parameter on the MQXCNVC call. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the specified character set, conversion must be carried out by the application.

MQRC_SOURCE_DECIMAL_ENC_ERROR
(2113, X'841') Packed-decimal encoding in message not recognized.

On an MQGET call with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the *Encoding* value in the message being retrieved specifies a decimal encoding that is not recognized. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

Corrective action: Check the decimal encoding that was specified when the message was put. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the required decimal encoding, conversion must be carried out by the application.

MQRC_SOURCE_FLOAT_ENC_ERROR
(2114, X'842') Floating-point encoding in message not recognized.

On an MQGET call, with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the *Encoding* value in the message being retrieved specifies a floating-point encoding that is not recognized. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

Corrective action: Check the floating-point encoding that was specified when the message was put. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the required floating-point encoding, conversion must be carried out by the application.

MQRC_SOURCE_INTEGER_ENC_ERROR
(2112, X'840') Source integer encoding not recognized.

On an MQGET call, with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the *Encoding* value in the message being retrieved specifies an integer encoding that is not recognized. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

If the message consists of several parts, each of which is described by its own *CodedCharSetId* and *Encoding* fields (for example, a message with format name MQFMT_DEAD_LETTER_HEADER), some parts may be converted

and other parts not converted. However, the values returned in the various *CodedCharSetId* and *Encoding* fields always correctly describe the relevant message data.

This reason code can also occur on the MQXCNVC call, when the *Options* parameter contains an unsupported MQDCC_SOURCE_★ value, or when MQDCC_SOURCE_ENC_UNDEFINED is specified for a UCS2 code page.

Corrective action: Check the integer encoding that was specified when the message was put. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the required integer encoding, conversion must be carried out by the application.

MQRC_SOURCE_LENGTH_ERROR
(2143, X'85F') Source length parameter not valid.

On the MQXCNVC call, the *SourceLength* parameter specifies a length that is less than zero or not consistent with the string's character set or content (for example, the character set is a double-byte character set, but the length is not a multiple of two). This reason also occurs if the *SourceLength* parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a length that is zero or greater.

MQRC_SRC_ENV_ERROR
(2261, X'8D5') Source environment data error.

This reason occurs when a channel exit that processes reference messages detects an error in the source environment data of a reference message header (MQRMH). One of the following is true:

- *SrcEnvLength* is less than zero.
- Source environment data is not present although *SrcEnvLength* is greater than zero.
- The range defined by *SrcEnvOffset* and *SrcEnvLength* is not wholly beyond the fixed fields in the MQRMH structure and within *StrucLength* bytes from the start of the structure.

The exit returns this reason in the *Feedback* field of the MQCXP structure. If an exception report is requested, it is copied to the *Feedback* field of the MQMD associated with the report.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify the source environment data correctly.

MQRC_SRC_NAME_ERROR
(2262, X'8D6') Source name data error.

This reason occurs when a channel exit that processes reference messages detects an error in the source name data of a reference message header (MQRMH). One of the following is true:

- *SrcNameLength* is less than zero.
- Source name data is not present although *SrcNameLength* is greater than zero.

- The range defined by *SrcNameOffset* and *SrcNameLength* is not wholly beyond the fixed fields in the MQRMH structure and within *StrucLength* bytes from the start of the structure.

The exit returns this reason in the *Feedback* field of the MQCXP structure. If an exception report is requested, it is copied to the *Feedback* field of the MQMD associated with the report.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Specify the source name data correctly.

MQRC_STORAGE_CLASS_ERROR
(2105, X'839') Storage class error.

The MQPUT or MQPUT1 call was issued, but the storage-class object defined for the queue does not exist.

This reason code occurs only on MVS/ESA.

Corrective action: Create the storage-class object required by the queue, or modify the queue definition to use an existing storage class. The name of the storage-class object used by the queue is given by the *StorageClass* queue attribute.

MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.

A call cannot complete because sufficient storage is not available to the queue manager.

Corrective action: Ensure that active applications are behaving correctly, for example, that they are not looping. If no problems are found, make more storage available.

On MVS/ESA, if no application problems are found, ask your systems programmer to increase the size of the region in which the queue manager runs.

MQRC_SUPPRESSED_BY_EXIT
(2109, X'83D') Call suppressed by exit program.

On any call other than MQCONN or MQDISC, the API crossing exit suppressed the call.

This reason code occurs only on MVS/ESA.

Corrective action: Obey the rules for API calls that the exit enforces. To find out the rules, see the writer of the exit.

MQRC_SYNCPOINT_LIMIT_REACHED
(2024, X'7E8') No more messages can be handled within current unit of work.

An MQGET, MQPUT, or MQPUT1 call failed because it would have caused the number of uncommitted messages in the current unit of work to exceed the limit defined for the queue manager (see the *MaxUncommittedMsgs* queue-manager attribute). The number of uncommitted messages is the sum of the following since the start of the current unit of work:

- Messages put by the application with the MQPMO_SYNCPOINT option

- Messages retrieved by the application with the MQGMO_SYNCPOINT option
- Trigger messages and COA report messages generated by the queue manager for messages put with the MQPMO_SYNCPOINT option
- COD report messages generated by the queue manager for messages retrieved with the MQGMO_SYNCPOINT option

Corrective action: Check whether the application is looping. If it is not, consider reducing the complexity of the application. Alternatively, increase the queue-manager limit for the maximum number of uncommitted messages within a unit of work:

- On MVS/ESA, the limit for the maximum number of uncommitted messages can be changed by using the DEFINE MAXSMSGS command.
- On OS/400, the limit for the maximum number of uncommitted messages can be changed by using the CHGMQM command.
- On Tandem NSK, the maximum number of I/O operations in a single TM/MP transaction has been exceeded. The application should cancel the transaction and retry with a smaller number of operations in the UOW. See the *MQSeries for Tandem NonStop Kernel System Management Guide* for more details.

MQRC_SYNCPOINT_NOT_AVAILABLE
(2072, X'818') Syncpoint support not available.

MQGMO_SYNCPOINT was specified on an MQGET call, or MQPMO_SYNCPOINT was specified on an MQPUT or MQPUT1 call, but the local queue manager was unable to honor the request. If the queue manager does not support units of work, the *SyncPoint* queue-manager attribute will have the value MQSP_NOT_AVAILABLE.

This reason code can also occur on the MQGET, MQPUT, and MQPUT1 calls when an external unit-of-work coordinator is being used. If that coordinator requires an explicit call to start the unit of work, but the application has not issued that call prior to the MQGET, MQPUT, or MQPUT1 call, reason code MQRC_SYNCPOINT_NOT_AVAILABLE is returned.

- On OS/400, this reason codes means that OS/400 Commitment Control is not started, or is unavailable for use by the queue manager.
- On MVS/ESA, this reason code does not occur.

Corrective action: Remove the specification of MQGMO_SYNCPOINT or MQPMO_SYNCPOINT, as appropriate.

On OS/400, if Commitment Control has not been started, start it. If this reason code occurs after Commitment Control has been started, contact your systems programmer.

MQRC_TARGET_BUFFER_ERROR
(2146, X'862') Target buffer parameter not valid.

On the MQXCNVC call, the *TargetBuffer* parameter pointer is not valid, or points to read-only storage, or to storage that cannot be accessed for the entire length specified by *TargetLength*. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a valid buffer.

MQRC_TARGET_CCSID_ERROR
(2115, X'843') Target coded character set identifier not valid.

The coded character-set identifier to which character data which is to be converted is not valid or not supported.

This can occur on the MQGET call when the MQGMO_CONVERT option is included in the *GetMsgOpts* parameter; the coded character-set identifier in error is the *CodedCharSetId* field in the *MsgDesc* parameter. In this case, the message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

This reason can also occur on the MQXCNVC call; the coded character-set identifier in error is the *TargetCCSID* parameter. Either the *TargetCCSID* parameter specifies a value which is not valid or not supported, or the *TargetCCSID* parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Check the character-set identifier that was specified for the *CodedCharSetId* field in the *MsgDesc* parameter on the MQGET call, or that was specified for the *SourceCCSID* parameter on the MQXCNVC call. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the specified character set, conversion must be carried out by the application.

MQRC_TARGET_DECIMAL_ENC_ERROR
(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

On an MQGET call with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the *Encoding* value in the *MsgDesc* parameter specifies a decimal encoding that is not recognized. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

Corrective action: Check the decimal encoding that was specified. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the required decimal encoding, conversion must be carried out by the application.

MQRC_TARGET_FLOAT_ENC_ERROR
(2118, X'846') Floating-point encoding specified by receiver not recognized.

On an MQGET call with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the *Encoding* value in the *MsgDesc* parameter specifies a floating-point encoding that is not recognized. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message returned, and the call completes with MQCC_WARNING.

Corrective action: Check the floating-point encoding that was specified. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the required floating-point encoding, conversion must be carried out by the application.

MQRC_TARGET_INTEGER_ENC_ERROR
(2116, X'844') Target integer encoding not recognized.

On an MQGET call with the MQGMO_CONVERT option included in the *GetMsgOpts* parameter, the *Encoding* value in the *MsgDesc* parameter specifies an integer encoding that is not recognized. The message data is returned unconverted, the values of the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to those of the message being retrieved, and the call completes with MQCC_WARNING.

This reason code can also occur on the MQXCNVC call, when the *Options* parameter contains an unsupported MQDCC_TARGET_⋆ value, or when MQDCC_TARGET_ENC_UNDEFINED is specified for a UCS2 code page.

Corrective action: Check the integer encoding that was specified. If this is correct, check that it is one for which queue-manager conversion is supported. If queue-manager conversion is not supported for the required integer encoding, conversion must be carried out by the application.

MQRC_TARGET_LENGTH_ERROR
(2144, X'860') Target length parameter not valid.

On the MQXCNVC call, the *TargetLength* parameter specifies a length that is less than zero. This reason also occurs if the *TargetLength* parameter pointer is not valid. (It is not always possible to detect an invalid parameter pointer; if not detected, unpredictable results occur.)

Corrective action: Specify a length that is zero or greater.

MQRC_TERMINATION_FAILED
(2287, X'8FF') Termination failed for an undefined reason.

This reason should be returned by an installable service component when the component is unable to complete termination successfully.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Correct the error and retry the operation.

MQRC_TM_ERROR
(2265, X'8D9') Trigger message structure not valid.

On an MQPUT or MQPUT1 call, the trigger message structure MQTM in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQTM structure. Ensure that the fields are set correctly.

MQRC_TMC_ERROR
(2191, X'88F') Character trigger message structure not valid.

On an MQPUT or MQPUT1 call, the character trigger message structure MQTMC or MQTMC2 in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQTMC or MQTMC2 structure. Ensure that the fields are set correctly.

MQRC_TRIGGER_CONTROL_ERROR
(2075, X'81B') Value for trigger-control attribute not valid.

On an MQSET call, the value specified for the MQIA_TRIGGER_CONTROL attribute selector is not valid.

Corrective action: Specify a valid value. See “Attributes for local queues and model queues” on page 348.

MQRC_TRIGGER_DEPTH_ERROR
(2076, X'81C') Value for trigger-depth attribute not valid.

On an MQSET call, the value specified for the MQIA_TRIGGER_DEPTH attribute selector is not valid.

Corrective action: Specify a value which is greater than zero. See “Attributes for local queues and model queues” on page 348.

MQRC_TRIGGER_MSG_PRIORITY_ERR
(2077, X'81D') Value for trigger-message-priority attribute not valid.

On an MQSET call, the value specified for the MQIA_TRIGGER_MSG_PRIORITY attribute selector is not valid.

Corrective action: Specify a value in the range 0 through the value of *MaxPriority* queue-manager attribute. See “Attributes for local queues and model queues” on page 348.

MQRC_TRIGGER_TYPE_ERROR
(2078, X'81E') Value for trigger-type attribute not valid.

On an MQSET call, the value specified for the MQIA_TRIGGER_TYPE attribute selector is not valid.

Corrective action: Specify a valid value. See “Attributes for local queues and model queues” on page 348.

MQRC_TRUNCATED_MSG_ACCEPTED
(2079, X'81F') Truncated message returned (processing completed).

On an MQGET call, the message length was too large to fit into the supplied buffer. The MQGMO_ACCEPT_TRUNCATED_MSG option was specified, so the call completes. The message is removed from the queue (subject to unit-of-work considerations), or, if this was a browse operation, the browse cursor is advanced to this message.

The *DataLength* parameter is set to the length of the message before truncation, the *Buffer* parameter contains as much of the message as fits, and the MQMD structure is filled in.

Corrective action: None, because the application expected this situation.

MQRC_TRUNCATED_MSG_FAILED
(2080, X'820') Truncated message returned (processing not completed).

On an MQGET call, the message length was too large to fit into the supplied buffer. The MQGMO_ACCEPT_TRUNCATED_MSG option was *not* specified, so the message has not been removed from the queue. If this was a browse operation, the browse cursor remains where it was before this call, but if MQGMO_BROWSE_FIRST was specified, the browse cursor is positioned logically before the highest-priority message on the queue.

The *DataLength* field is set to the length of the message before truncation, the *Buffer* parameter contains as much of the message as fits, and the MQMD structure is filled in.

Corrective action: Supply a buffer that is at least as large as *DataLength*, or specify MQGMO_ACCEPT_TRUNCATED_MSG if not all of the message data is required.

MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

The call was rejected because an unexpected error occurred.

Corrective Action: Check the application's parameter list to ensure, for example, that the correct number of parameters was passed, and that data pointers and storage keys are valid. If the problem cannot be resolved, contact your system programmer.

- On MVS/ESA, check whether any information has been displayed on the console. If this error occurs on an MQCONN call, check that the subsystem named is an active MQ subsystem. In particular, check that it is not a DB2 subsystem. If the problem cannot be resolved, rerun the application with a CSQSNAP DD card (if you have not already got a dump) and send the resulting dump to IBM.
- On OS/2 and OS/400, consult the FFST record to obtain more detail about the problem.
- On OpenVMS, Tandem NSK, and UNIX systems, consult the FDC file to obtain more detail about the problem.

MQRC_UNKNOWN_ALIAS_BASE_Q
(2082, X'822') Unknown alias base queue.

An MQOPEN or MQPUT1 call 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.

MQRC_UNKNOWN_AUTH_ENTITY
(2293, X'8F5') Authorization entity unknown to service.

This reason should be returned by the authority installable service component when the name specified by the *AuthEntityName* parameter is not recognized.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Ensure that the entity is defined.

MQRC_UNKNOWN_DEF_XMIT_Q
(2197, X'895') Unknown default transmission queue.

An MQOPEN or MQPUT1 call was issued specifying a remote queue as the destination. If a local definition of the remote queue was specified, or if a queue-manager alias is being resolved, the *XmitQName* attribute in the local definition is blank.

Because there is no queue defined with the same name as the destination queue manager, the queue manager has attempted to use the default transmission queue. However, the name defined by the *DefXmitQName* queue-manager attribute is not the name of a locally-defined queue.

Corrective Action: Correct the queue definitions, or the queue-manager attribute. See the *MQSeries Application Programming Guide* for more information.

MQRC_UNKNOWN_ENTITY
(2292, X'8F4') Entity unknown to service.

This reason should be returned by the authority installable service component when the name specified by the *EntityName* parameter is not recognized.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Ensure that the entity is defined.

MQRC_UNKNOWN_OBJECT_NAME
(2085, X'825') Unknown object name.

On an MQOPEN or MQPUT1 call, the *ObjectQMgrName* field in the object descriptor MQOD is set to one of the following:

- Blank
- The name of the local queue manager
- The name of a local definition of a remote queue (a queue-manager alias) in which the *RemoteQMgrName* attribute is the name of the local queue manager

However, the *ObjectName* field in the object descriptor is not recognized for the specified object type.

See also MQRC_Q_DELETED.

Corrective action: Specify a valid object name. Ensure that the name is padded to the right with blanks if necessary. If this is correct, check the queue definitions.

MQRC_UNKNOWN_OBJECT_Q_MGR
(2086, X'826') Unknown object queue manager.

On an MQOPEN or MQPUT1 call, the *ObjectQMgrName* field in the object descriptor MQOD does not satisfy the naming rules for objects. For more information, see the *MQSeries Application Programming Guide*.

This reason also occurs if the *ObjectType* field in the object descriptor has the value MQOT_Q_MGR, and the *ObjectQMgrName* field is not blank, but the name specified is not the name of the local queue manager.

Corrective Action: Specify a valid queue manager name (or all blanks or an initial null character to refer to the local queue manager). Ensure that the name is padded to the right with blanks or terminated with a null character if necessary.

MQRC_UNKNOWN_Q_NAME
(2288, X'8F0') Queue name not found.

This reason should be returned by the MQZ_LOOKUP_NAME installable service component when the name specified for the *QName* parameter is not recognized.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: None. See the *MQSeries Programmable System Management* book for details of this call.

MQRC_UNKNOWN_REF_OBJECT
(2294, X'8F6') Reference object unknown.

This reason should be returned by the MQZ_COPY_ALL_AUTHORITY installable service component when the name specified by the *RefObjectName* parameter is not recognized.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: Ensure that the reference object is defined. See the *MQSeries Programmable System Management* book for details of this call.

MQRC_UNKNOWN_REMOTE_Q_MGR
(2087, X'827') Unknown remote queue manager.

On an MQOPEN or MQPUT1 call, an error occurred with the queue-name resolution, for one of the following reasons:

- *ObjectQMgrName* is either blank or the name of the local queue manager, and *ObjectName* is the name of a local definition of a remote queue, which has a blank *XmitQName*. However, there is no (transmission) queue defined with the name of *RemoteQMgrName*, and the *DefXmitQName* queue-manager attribute is blank.
- *ObjectQMgrName* is the name of a queue-manager alias definition (held as the local definition of a remote queue), which has a blank *XmitQName*. However, there is no (transmission) queue defined with the name of *RemoteQMgrName*, and the *DefXmitQName* queue-manager attribute is blank.
- *ObjectQMgrName* specified is not:
  - Blank
  - The name of the local queue manager
  - The name of a local queue
  - The name of a queue-manager alias definition (that is, a local definition of a remote queue with a blank *RemoteQName*)

  and the *DefXmitQName* queue-manager attribute is blank.
- *ObjectQMgrName* is blank or is the name of the local queue manager, and *ObjectName* is the name of a local definition of a remote queue (or an alias to one), for which *RemoteQMgrName* is either blank or is the name of the local queue manager. Note that this error occurs even if the *XmitQName* is not blank.
- *ObjectQMgrName* is the name of a local definition of a remote queue. In this context, this should be a queue-manager alias definition, but the *RemoteQName* in the definition is not blank.
- *ObjectQMgrName* is the name of a model queue.
- The queue name is resolved through a cell directory. However, there is no queue defined with the same name as the remote queue manager name obtained from the cell directory. Also, the *DefXmitQName* queue-manager attribute is blank.

Corrective action: Check the values specified for *ObjectQMgrName* and *ObjectName*. If these are correct, check the queue definitions.

MQRC_UNKNOWN_REPORT_OPTION
(2104, X'838') Report option(s) in message descriptor not recognized.

An MQPUT or MQPUT1 call was issued, but the *Report* field in the message

descriptor MQMD contains one or more options which are not recognized by the local queue manager. The options are accepted.

The options that cause this reason code to be returned depend on the destination of the message; see Appendix C, “Report options and message flags” on page 489 for more details.

Corrective action: If this reason code is expected, no corrective action is required.

If this reason code is not expected, do the following:

1. Ensure that the *Report* field in the message descriptor is initialized with a value when the message descriptor is declared, or is assigned a value prior to the MQPUT or MQPUT1 call.
2. Ensure that the report options specified are ones which are documented in this book; see the *Report* field described in “MQMD – Message descriptor” on page 98 for valid report options. Remove any report options which are not documented in this book.
3. If multiple report options are being set by adding the individual report options together, ensure that the same report option is not added twice.
4. Check that conflicting report options are not specified. For example, do not add both MQRO_EXCEPTION and MQRO_EXCEPTION_WITH_DATA to the *Report* field; only one of these can be specified.

MQRC_UNKNOWN_XMIT_Q
(2196, X'894') Unknown transmission queue.

On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. The *ObjectName* or the *ObjectQMgrName* in the object descriptor specifies the name of a local definition of a remote queue (in the latter case queue-manager aliasing is being used), but the *XmitQName* attribute of the definition is not blank and not the name of a locally-defined queue.

Corrective action: Check the values specified for *ObjectName* and *ObjectQMgrName*. If these are correct, check the queue definitions. For more information on transmission queues, see the *MQSeries Application Programming Guide*.

MQRC_UOW_CANCELED
(2297, X'8F9') Unit of work (TM/MP transaction) has been cancelled. This may have been performed by TM/MP itself (there are some system wide TM/MP configuration parameters controlling long running transactions and audit trail sizes) or by the application program issuing an ABORT_TRANSACTION. All updates performed to MQSeries resources are backed out.

MQRC_UOW_IN_PROGRESS
(2128, X'850') Unit of work already started.

An MQBEGIN call was issued to start a unit of work coordinated by the queue manager, but a unit of work is already in existence for the connection handle specified. This may be a global unit of work started by a previous MQBEGIN call, or a unit of work that is local to the queue manager or one of the cooperating resource managers. No more than one unit of work can exist concurrently for a connection handle.

This reason code occurs in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Corrective action: Review the application logic to determine why there is a unit of work already in existence. Move the MQBEGIN call to the appropriate place in the application.

MQRC_UOW_NOT_AVAILABLE
(2255, X'8CF') Unit of work not available for the queue manager to use.

An MQGET, MQPUT, or MQPUT1 call was issued to get or put a message outside a unit of work, but the options specified on the call required the queue manager to process the call within a unit of work. Because there is already a user-defined unit of work in existence, the queue manager was unable to create a temporary unit of work for the duration of the call.

This reason occurs in the following circumstances:

- On an MQGET call, when the MQGMO_COMPLETE_MSG option is specified in MQGMO and the logical message to be retrieved is persistent and consists of two or more segments.
- On an MQPUT or MQPUT1 call, when the MQMF_SEGMENTATION_ALLOWED flag is specified in MQMD and the message requires segmentation.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Issue the MQGET, MQPUT, or MQPUT1 call inside the user-defined unit of work. Alternatively, for the MQPUT or MQPUT1 call, reduce the size of the message so that it does not require segmentation by the queue manager.

MQRC_UNIT_OF_WORK_NOT_STARTED
(2232, X'8B8') An attempt has been made to perform an MQPUT or MQGET call with the SYNCPOINT option specified (the default), but no TM/MP transaction has been started. Ensure a TM/MP transaction is available, or issue the MQPUT or MQGET call with the NO_SYNCPOINT option, in which case a transaction is started automatically.

MQRC_USER_ID_NOT_AVAILABLE
(2291, X'8F3') Unable to determine the user ID.

This reason should be returned by the MQZ_FIND_USERID installable service component when the user ID cannot be determined.

On MVS/ESA and OS/400, this reason code does not occur.

Corrective action: None. See the *MQSeries Programmable System Management* book for details of this call.

MQRC_WAIT_INTERVAL_ERROR
(2090, X'82A') Wait interval in MQGMO not valid.

On the MQGET call, the value specified for the *WaitInterval* field in the *GetMsgOpts* parameter is not valid.

Corrective action: Specify a value greater than or equal to zero, or the special value MQWI_UNLIMITED if an indefinite wait is required.

MQRC_WRONG_GMO_VERSION
(2256, X'8D0') Wrong version of MQGMO supplied.

An MQGET call was issued specifying options that required an MQGMO with a version number not less than MQGMO_VERSION_2, but the MQGMO supplied did not satisfy this condition.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Modify the program to pass a version-2 MQGMO. Check the program logic to ensure that the *Version* field in MQGMO has been set to MQGMO_VERSION_2. Alternatively, remove the option that requires the version-2 MQGMO.

MQRC_WRONG_MD_VERSION
(2257, X'8D1') Wrong version of MQMD supplied.

An MQGET, MQPUT, or MQPUT1 call was issued specifying options that required an MQMD with a version number not less than MQMD_VERSION_2, but the MQMD supplied did not satisfy this condition.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Modify the program to pass a version-2 MQMD. Check the program logic to ensure that the *Version* field in MQMD has been set to MQMD_VERSION_2. Alternatively, remove the option that requires the version-2 MQMD.

MQRC_XMIT_Q_TYPE_ERROR
(2091, X'82B') Transmission queue not local.

On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. The *ObjectName* or *ObjectQMgrName* field in the object descriptor specifies the name of a local definition of a remote queue but one of the following applies to the *XmitQName* attribute of the definition:

- *XmitQName* is not blank, but specifies a queue that is not a local queue
- *XmitQName* is blank, but *RemoteQMgrName* specifies a queue that is not a local queue

This reason also occurs if the queue name is resolved through a cell directory, and the remote queue manager name obtained from the cell directory is the name of a queue, but this is not a local queue.

Corrective action: Check the values specified for *ObjectName* and *ObjectQMgrName*. If these are correct, check the queue definitions. For more information on transmission queues, see the *MQSeries Application Programming Guide*.

MQRC_XMIT_Q_USAGE_ERROR
(2092, X'82C') Transmission queue with wrong usage.

On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager, but one of the following occurred:

- *ObjectQMgrName* specifies the name of a local queue, but it does not have a *Usage* attribute of MQUS_TRANSMISSION.

- The *ObjectName* or *ObjectQMgrName* field in the object descriptor specifies the name of a local definition of a remote queue but one of the following applies to the *XmitQName* attribute of the definition:
  - *XmitQName* is not blank, but specifies a queue that does not have a *Usage* attribute of MQUS_TRANSMISSION
  - *XmitQName* is blank, but *RemoteQMgrName* specifies a queue that does not have a *Usage* attribute of MQUS_TRANSMISSION
- The queue name is resolved through a cell directory, and the remote queue manager name obtained from the cell directory is the name of a local queue, but it does not have a *Usage* attribute of MQUS_TRANSMISSION.

Corrective action: Check the values specified for *ObjectName* and *ObjectQMgrName*. If these are correct, check the queue definitions. For more information on transmission queues, see the *MQSeries Application Programming Guide*.

MQRC_XQH_ERROR
(2260, X'8D4') Transmission queue header structure not valid.

On an MQPUT or MQPUT1 call, the transmission queue header structure MQXQH in the message data is not valid.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQXQH structure. Ensure that the fields are set correctly.

MQRC_XWAIT_CANCELED
(2107, X'83B') MQXWAIT call canceled.

An MQXWAIT call has been canceled because a STOP CHINIT command has been issued (or the queue manager has been stopped, which causes the same effect). Refer to the *MQSeries Intercommunication* book for details of the MQXWAIT call.

This reason code occurs only on MVS/ESA.

Corrective action: Tidy up and terminate.

MQRC_XWAIT_ERROR
(2108, X'83C') Invocation of MQXWAIT call not valid.

An MQXWAIT call was issued, but the invocation was not valid for one of the following reasons:

- The wait descriptor MQXWD contains data which is not valid.
- The linkage stack level is not valid.
- The addressing mode is not valid.
- There are too many wait events outstanding.

This reason code occurs only on MVS/ESA.

Corrective action: Obey the rules for using the MQXWAIT call. Refer to the *MQSeries Intercommunication* book for details of this call.

Chapter 6. MQSeries constants

This chapter specifies the values of all of the named constants that are mentioned in this book. For other MQI constants, refer to the *MQSeries Intercommunication* book and the *MQSeries Programmable System Management* book.

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

Notes:

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

List of constants

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

MQ_★ (Lengths of character string and byte fields)

See, for example, the *CharAttrs* parameter described in "MQINQ – Inquire about object attributes" on page 285 and "MQSET – Set object attributes" on page 333.

MQ_ABEND_CODE_LENGTH	4	X'00000004'
MQ_ACCOUNTING_TOKEN_LENGTH	32	X'00000020'
MQ_APPL_IDENTITY_DATA_LENGTH	32	X'00000020'
MQ_APPL_NAME_LENGTH	28	X'0000001C'
MQ_APPL_ORIGIN_DATA_LENGTH	4	X'00000004'
MQ_ATTENTION_ID_LENGTH	4	X'00000004'
MQ_AUTHENTICATOR_LENGTH	8	X'00000008'
MQ_BRIDGE_NAME_LENGTH	24	X'00000018'
MQ_CANCEL_CODE_LENGTH	4	X'00000004'
MQ_CHANNEL_DATE_LENGTH	12	X'0000000C'
MQ_CHANNEL_DESC_LENGTH	64	X'00000040'
MQ_CHANNEL_NAME_LENGTH	20	X'00000014'
MQ_CHANNEL_TIME_LENGTH	8	X'00000008'
MQ_CONN_NAME_LENGTH	264	X'00000108'
MQ_CORREL_ID_LENGTH	24	X'00000018'
MQ_CREATION_DATE_LENGTH	12	X'0000000C'
MQ_CREATION_TIME_LENGTH	8	X'00000008'
MQ_EXIT_DATA_LENGTH	32	X'00000020'

MQ_EXIT_NAME_LENGTH	(environment specific)
MQ_EXIT_USER_AREA_LENGTH	16 X'00000010'
MQ_FACILITY_LENGTH	8 X'00000008'
MQ_FACILITY_LIKE_LENGTH	4 X'00000004'
MQ_FORMAT_LENGTH	8 X'00000008'
MQ_FUNCTION_LENGTH	4 X'00000004'
MQ_GROUP_ID_LENGTH	24 X'00000018'
MQ_LTERM_OVERRIDE_LENGTH	8 X'00000008'
MQ_LUWID_LENGTH	16 X'00000010'
MQ_MCA_JOB_NAME_LENGTH	28 X'0000001C'
MQ_MCA_NAME_LENGTH	20 X'00000014'
MQ_MFS_MAP_NAME_LENGTH	8 X'00000008'
MQ_MODE_NAME_LENGTH	8 X'00000008'
MQ_MSG_HEADER_LENGTH	4000 X'00000FA0'
MQ_MSG_ID_LENGTH	24 X'00000018'
MQ_NAMELIST_DESC_LENGTH	64 X'00000040'
MQ_NAMELIST_NAME_LENGTH	48 X'00000030'
MQ_OBJECT_INSTANCE_ID_LENGTH	24 X'00000018'
MQ_PASSWORD_LENGTH	12 X'0000000C'
MQ_PROCESS_APPL_ID_LENGTH	256 X'00000100'
MQ_PROCESS_DESC_LENGTH	64 X'00000040'
MQ_PROCESS_ENV_DATA_LENGTH	128 X'00000080'
MQ_PROCESS_NAME_LENGTH	48 X'00000030'
MQ_PROCESS_USER_DATA_LENGTH	128 X'00000080'
MQ_PUT_APPL_NAME_LENGTH	28 X'0000001C'
MQ_PUT_DATE_LENGTH	8 X'00000008'
MQ_PUT_TIME_LENGTH	8 X'00000008'
MQ_Q_DESC_LENGTH	64 X'00000040'
MQ_Q_MGR_DESC_LENGTH	64 X'00000040'
MQ_Q_MGR_NAME_LENGTH	48 X'00000030'
MQ_Q_NAME_LENGTH	48 X'00000030'
MQ_REMOTE_SYS_ID_LENGTH	4 X'00000004'
MQ_SHORT_CONN_NAME_LENGTH	20 X'00000014'
MQ_START_CODE_LENGTH	4 X'00000004'
MQ_STORAGE_CLASS_LENGTH	8 X'00000008'
MQ_TOTAL_EXIT_DATA_LENGTH	999 X'000003E7'
MQ_TOTAL_EXIT_NAME_LENGTH	999 X'000003E7'
MQ_TP_NAME_LENGTH	64 X'00000040'
MQ_TRAN_INSTANCE_ID_LENGTH	16 X'00000010'
MQ_TRANSACTION_ID_LENGTH	4 X'00000004'
MQ_TRIGGER_DATA_LENGTH	64 X'00000040'
MQ_USER_ID_LENGTH	12 X'0000000C'

MQACT_⋆ (Accounting token)

See the *AccountingToken* field described in “MQMD – Message descriptor” on page 98.

MQACT_NONE	X'00...00' (32 nulls)

For the C programming language, the following is also defined:

MQACT_NONE_ARRAY	'\0','\0',...'\0','\0'

MQAT_★ (Application type)

See the *PutApplType* field described in "MQMD – Message descriptor" on page 98, and the *ApplType* attribute described in "Attributes for process definitions" on page 367.

MQAT_UNKNOWN	-1	X'FFFFFFFF'
MQAT_NO_CONTEXT	0	X'00000000'
MQAT_CICS	1	X'00000001'
MQAT_MVS	2	X'00000002'
MQAT_IMS	3	X'00000003'
MQAT_OS2	4	X'00000004'
MQAT_DOS	5	X'00000005'
MQAT_AIX	6	X'00000006'
MQAT_UNIX	6	X'00000006'
MQAT_QMGR	7	X'00000007'
MQAT_OS400	8	X'00000008'
MQAT_WINDOWS	9	X'00000009'
MQAT_CICS_VSE	10	X'0000000A'
MQAT_WINDOWS_NT	11	X'0000000B'
MQAT_VMS	12	X'0000000C'
MQAT_GUARDIAN	13	X'0000000D'
MQAT_NSK	13	X'0000000D'
MQAT_VOS	14	X'0000000E'
MQAT_IMS_BRIDGE	19	X'00000013'
MQAT_XCF	20	X'00000014'
MQAT_CICS_BRIDGE	21	X'00000015'
MQAT_USER_FIRST	65536	X'00010000'
MQAT_USER_LAST	999999999	X'3B9AC9FF'
MQAT_DEFAULT	(environment specific)	

MQBO_★ (Begin options)

See the *Options* field described in "MQBO – Begin options" on page 19.

MQBO_NONE	0	X'00000000'

MQBO_★ (Begin options structure identifier)

See the *StrucId* field described in "MQBO – Begin options" on page 19.

MQBO_STRUC_ID	'BObb'

For the C programming language, the following is also defined:

MQBO_STRUC_ID_ARRAY	'B','O','b','b'

MQBO_★ (Begin options version)

See the *Version* field described in "MQBO – Begin options" on page 19.

MQBO_VERSION_1	1	X'00000001'
MQBO_CURRENT_VERSION	1	X'00000001'

MQCA_★ (Character attribute selector)

See the *Selectors* parameter described in "MQINQ – Inquire about object attributes" on page 285 and "MQSET – Set object attributes" on page 333.

MQCA_FIRST	2001	X'000007D1'
MQCA_APPL_ID	2001	X'000007D1'
MQCA_BASE_Q_NAME	2002	X'000007D2'
MQCA_COMMAND_INPUT_Q_NAME	2003	X'000007D3'
MQCA_CREATION_DATE	2004	X'000007D4'
MQCA_CREATION_TIME	2005	X'000007D5'
MQCA_DEAD_LETTER_Q_NAME	2006	X'000007D6'
MQCA_ENV_DATA	2007	X'000007D7'
MQCA_INITIATION_Q_NAME	2008	X'000007D8'
MQCA_NAMELIST_DESC	2009	X'000007D9'
MQCA_NAMELIST_NAME	2010	X'000007DA'
MQCA_PROCESS_DESC	2011	X'000007DB'
MQCA_PROCESS_NAME	2012	X'000007DC'
MQCA_Q_DESC	2013	X'000007DD'
MQCA_Q_MGR_DESC	2014	X'000007DE'
MQCA_Q_MGR_NAME	2015	X'000007DF'
MQCA_Q_NAME	2016	X'000007E0'
MQCA_REMOTE_Q_MGR_NAME	2017	X'000007E1'
MQCA_REMOTE_Q_NAME	2018	X'000007E2'
MQCA_BACKOUT_REQ_Q_NAME	2019	X'000007E3'
MQCA_NAMES	2020	X'000007E4'
MQCA_USER_DATA	2021	X'000007E5'
MQCA_STORAGE_CLASS	2022	X'000007E6'
MQCA_TRIGGER_DATA	2023	X'000007E7'
MQCA_XMIT_Q_NAME	2024	X'000007E8'
MQCA_DEF_XMIT_Q_NAME	2025	X'000007E9'
MQCA_CHANNEL_AUTO_DEF_EXIT	2026	X'000007EA'
MQCA_LAST	4000	X'00000FA0'
MQCA_LAST_USED	(environment specific)	

MQCC_★ (Completion code)

See the *CompCode* parameter.

MQCC_UNKNOWN	-1	X'FFFFFFFF'
MQCC_OK	0	X'00000000'
MQCC_WARNING	1	X'00000001'
MQCC_FAILED	2	X'00000002'

MQCCSI_★ (Coded character set identifier)

See the *CodedCharSetId* field described in "MQMD – Message descriptor" on page 98.

MQCCSI_EMBEDDED	-1	X'FFFFFFFF'
MQCCSI_DEFAULT	0	X'00000000'
MQCCSI_Q_MGR	0	X'00000000'

MQCFUNC_★ (CICS header function name)

See the *Function* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCFUNC_MQCONN	'CONN'
MQCFUNC_MQGET	'GETb'
MQCFUNC_MQINQ	'INQb'
MQCFUNC_MQOPEN	'OPEN'
MQCFUNC_MQPUT	'PUTb'
MQCFUNC_MQPUT1	'PUT1'
MQCFUNC_NONE	'bbbb'

For the C programming language, the following is also defined:

MQCFUNC_MQCONN_ARRAY	'C','O','N','N'
MQCFUNC_MQGET_ARRAY	'G','E','T','b'
MQCFUNC_MQINQ_ARRAY	'I','N','Q','b'
MQCFUNC_MQOPEN_ARRAY	'O','P','E','N'
MQCFUNC_MQPUT_ARRAY	'P','U','T','b'
MQCFUNC_MQPUT1_ARRAY	'P','U','T','1'
MQCFUNC_NONE_ARRAY	'b','b','b','b'

MQCGWI_★ (CICS header get-wait interval)

See the *GetWaitInterval* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCGWI_DEFAULT	-2	X'FFFFFFFE'

MQCI_★ (Correlation identifier)

See the *CorrelId* field described in "MQMD – Message descriptor" on page 98.

MQCI_NONE	X'00...00' (24 nulls)
MQCI_NEW_SESSION	X'414D51214E45575F534553...'

For the C programming language, the following is also defined:

MQCI_NONE_ARRAY	'\0','\0',...'\0','\0'
MQCI_NEW_SESSION_ARRAY	'\x41','\x4d','\x51',...

MQCIH_★ (CICS header flags)

See the *Flags* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCIH_NONE	0	X'00000000'

MQCIH_★ (CICS header length)

See the *StrucLength* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCIH_LENGTH_1	164	X'000000A4'

MQCIH_★ (CICS header structure identifier)

See the *StrucId* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCIH_STRUC_ID	'CIHb'

For the C programming language, the following is also defined:

MQCIH_STRUC_ID_ARRAY	'C','I','H','b'

MQCIH_★ (CICS header version)

See the *Version* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCIH_VERSION_1	1	X'00000001'
MQCIH_CURRENT_VERSION	1	X'00000001'

MQCLT_★ (CICS header link type)

See the *LinkType* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCLT_PROGRAM	1	X'00000001'

MQCMDL_★ (Command level)

See the *CommandLevel* attribute described in "Attributes for the queue manager" on page 370.

MQCMDL_LEVEL_1	100	X'00000064'
MQCMDL_LEVEL_101	101	X'00000065'
MQCMDL_LEVEL_110	110	X'0000006E'
MQCMDL_LEVEL_114	114	X'00000072'
MQCMDL_LEVEL_120	120	X'00000078'
MQCMDL_LEVEL_200	200	X'000000C8'
MQCMDL_LEVEL_201	201	X'000000C9'
MQCMDL_LEVEL_220	220	X'000000DC'
MQCMDL_LEVEL_221	221	X'000000DD'
MQCMDL_LEVEL_320	320	X'00000140'
MQCMDL_LEVEL_420	420	X'000001A4'
MQCMDL_LEVEL_500	500	X'000001F4'

MQCNO_★ (Connect options)

See the *Options* field described in "MQCNO – Connect options" on page 35.

MQCNO_STANDARD_BINDING	0	X'00000000'
MQCNO_NONE	0	X'00000000'
MQCNO_FASTPATH_BINDING	1	X'00000001'

MQCNO_★ (Connect options structure identifier)

See the *StrucId* field described in "MQCNO – Connect options" on page 35.

MQCNO_STRUC_ID	'CNOb'

For the C programming language, the following is also defined:

MQCNO_STRUC_ID_ARRAY	'C','N','O','b'

MQCNO_★ (Connect options version)

See the *Version* field described in "MQCNO – Connect options" on page 35.

MQCNO_VERSION_1	1	X'00000001'
MQCNO_CURRENT_VERSION	1	X'00000001'

MQCO_★ (Close options)

See the *Options* parameter described in "MQCLOSE – Close object" on page 248.

MQCO_NONE	0	X'00000000'
MQCO_DELETE	1	X'00000001'
MQCO_DELETE_PURGE	2	X'00000002'

MQCODL_★ (CICS header output data length)

See the *OutputDataLength* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCODL_AS_INPUT	-1	X'FFFFFFFF'

MQCRC_★ (CICS header return code)

See the *ReturnCode* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCRC_OK	0	X'00000000'
MQCRC_CICS_EXEC_ERROR	1	X'00000001'
MQCRC_MQ_API_ERROR	2	X'00000002'
MQCRC_BRIDGE_ERROR	3	X'00000003'
MQCRC_BRIDGE_ABEND	4	X'00000004'
MQCRC_APPLICATION_ABEND	5	X'00000005'
MQCRC_SECURITY_ERROR	6	X'00000006'
MQCRC_PROGRAM_NOT_AVAILABLE	7	X'00000007'
MQCRC_BRIDGE_TIMEOUT	8	X'00000008'
MQCRC_TRANSID_NOT_AVAILABLE	9	X'00000009'

MQCUOWC_★ (CICS header unit-of-work control)

See the *UOWControl* field described in "MQCIH – CICS bridge header (MVS/ESA only)" on page 21.

MQCUOWC_MIDDLE	16	X'00000010'
MQCUOWC_FIRST	17	X'00000011'
MQCUOWC_COMMIT	256	X'00000100'
MQCUOWC_LAST	272	X'00000110'

MQCUOWC_ONLY	273	X'00000111'
MQCUOWC_BACKOUT	4352	X'00001100'

MQDCC_★ (Convert-characters masks and factors)

See the *Options* parameter described in "MQXCNVC – Convert characters" on page 509.

MQDCC_SOURCE_ENC_MASK	240	X'000000F0'
MQDCC_TARGET_ENC_MASK	3840	X'00000F00'
MQDCC_SOURCE_ENC_FACTOR	16	X'00000010'
MQDCC_TARGET_ENC_FACTOR	256	X'00000100'

MQDCC_★ (Convert-characters options)

See the *Options* parameter described in "MQXCNVC – Convert characters" on page 509.

MQDCC_SOURCE_ENC_UNDEFINED	0	X'00000000'
MQDCC_TARGET_ENC_UNDEFINED	0	X'00000000'
MQDCC_NONE	0	X'00000000'
MQDCC_DEFAULT_CONVERSION	1	X'00000001'
MQDCC_SOURCE_ENC_NORMAL	16	X'00000010'
MQDCC_SOURCE_ENC_REVERSED	32	X'00000020'
MQDCC_TARGET_ENC_NORMAL	256	X'00000100'
MQDCC_TARGET_ENC_REVERSED	512	X'00000200'
MQDCC_SOURCE_ENC_NATIVE	(environment specific)	
MQDCC_TARGET_ENC_NATIVE	(environment specific)	

MQDH_★ (Distribution header structure identifier)

See the *StrucId* field described in "MQDH – Distribution header" on page 39.

MQDH_STRUC_ID	'DHbb'

For the C programming language, the following is also defined:

MQDH_STRUC_ID_ARRAY	'D','H','b','b'

MQDH_★ (Distribution header version)

See the *Version* field described in "MQDH – Distribution header" on page 39.

MQDH_VERSION_1	1	X'00000001'
MQDH_CURRENT_VERSION	1	X'00000001'

MQDHF_★ (Distribution header flags)

See the *Flags* field described in "MQDH – Distribution header" on page 39.

MQDHF_NONE	0	X'00000000'
MQDHF_NEW_MSG_IDS	1	X'00000001'

MQDL_★ (Distribution list support)

See the *DistLists* attributes described in "Attributes for the queue manager" on page 370 and "Attributes for local queues and model queues" on page 348.

MQDL_NOT_SUPPORTED	0	X'00000000'
MQDL_SUPPORTED	1	X'00000001'

MQDLH_★ (Dead-letter header structure identifier)

See the *StrucId* field described in "MQDLH – Dead-letter header" on page 45.

MQDLH_STRUC_ID	'DLHb'

For the C programming language, the following is also defined:

MQDLH_STRUC_ID_ARRAY	'D','L','H','b'

MQDLH_★ (Dead-letter header version)

See the *Version* field described in "MQDLH – Dead-letter header" on page 45.

MQDLH_VERSION_1	1	X'00000001'
MQDLH_CURRENT_VERSION	1	X'00000001'

MQDXP_★ (Data-conversion-exit parameter structure identifier)

See the *StrucId* field described in "MQDXP – Data-conversion exit parameter structure" on page 502.

MQDXP_STRUC_ID	'DXPb'

For the C programming language, the following is also defined:

MQDXP_STRUC_ID_ARRAY	'D','X','P','b'

MQDXP_★ (Data-conversion-exit parameter structure version)

See the *Version* field described in "MQDXP – Data-conversion exit parameter structure" on page 502.

MQDXP_VERSION_1	1	X'00000001'
MQDXP_CURRENT_VERSION	1	X'00000001'

MQEC_★ (Signal event-control-block completion code)

See the *Signal1* field described in "MQGMO – Get-message options" on page 56.

MQEC_MSG_ARRIVED	2	X'00000002'
MQEC_WAIT_INTERVAL_EXPIRED	3	X'00000003'
MQEC_WAIT_CANCELED	4	X'00000004'
MQEC_Q_MGR_QUIESCING	5	X'00000005'
MQEC_CONNECTION_QUIESCING	6	X'00000006'

MQEI_★ (Expiry interval)

See the *Expiry* field described in "MQMD – Message descriptor" on page 98.

MQEI_UNLIMITED	-1	X'FFFFFFFF'

MQENC_★ (Encoding)

See the *Encoding* field described in "MQMD – Message descriptor" on page 98.

MQENC_NATIVE	(environment specific)

This constant has the following values in the environments indicated:

OS/2, DOS client, Windows client	546
16-bit Windows, 32-bit Windows, Windows NT	546
Micro Focus COBOL on OS/2 and Windows NT	17
OpenVMS	273
MVS/ESA	785
OS/400	273
Tandem NSK	273
UNIX systems (AIX, AT&T, HP-UX)	273

MQENC_★ (Encoding masks)

See Appendix B, "Machine encodings" on page 485.

MQENC_INTEGER_MASK	15	X'0000000F'
MQENC_DECIMAL_MASK	240	X'000000F0'
MQENC_FLOAT_MASK	3840	X'00000F00'
MQENC_RESERVED_MASK	-4096	X'FFFFF000'

MQENC_★ (Encoding for packed-decimal integers)

See Appendix B, "Machine encodings" on page 485.

MQENC_DECIMAL_UNDEFINED	0	X'00000000'
MQENC_DECIMAL_NORMAL	16	X'00000010'
MQENC_DECIMAL_REVERSED	32	X'00000020'

MQENC_★ (Encoding for floating-point numbers)

See Appendix B, "Machine encodings" on page 485.

MQENC_FLOAT_UNDEFINED	0	X'00000000'
MQENC_FLOAT_IEEE_NORMAL	256	X'00000100'
MQENC_FLOAT_IEEE_REVERSED	512	X'00000200'
MQENC_FLOAT_S390	768	X'00000300'

MQENC_★ (Encoding for binary integers)

See Appendix B, "Machine encodings" on page 485.

MQENC_INTEGER_UNDEFINED	0	X'00000000'
MQENC_INTEGER_NORMAL	1	X'00000001'
MQENC_INTEGER_REVERSED	2	X'00000002'

MQEVR_★ (Event reporting)

MQEVR_DISABLED	0	X'00000000'
MQEVR_ENABLED	1	X'00000001'

MQFB_★ (Feedback)

See the *Feedback* field described in “MQMD – Message descriptor” on page 98, and the *Reason* field described in “MQDLH – Dead-letter header” on page 45; see also the MQRC_★ values.

MQFB_NONE	0	X'00000000'
MQFB_SYSTEM_FIRST	1	X'00000001'
MQFB_QUIT	256	X'00000100'
MQFB_EXPIRATION	258	X'00000102'
MQFB_COA	259	X'00000103'
MQFB_COD	260	X'00000104'
MQFB_CHANNEL_COMPLETED	262	X'00000106'
MQFB_CHANNEL_FAIL_RETRY	263	X'00000107'
MQFB_CHANNEL_FAIL	264	X'00000108'
MQFB_APPL_CANNOT_BE_STARTED	265	X'00000109'
MQFB_TM_ERROR	266	X'0000010A'
MQFB_APPL_TYPE_ERROR	267	X'0000010B'
MQFB_STOPPED_BY_MSG_EXIT	268	X'0000010C'
MQFB_XMIT_Q_MSG_ERROR	271	X'0000010F'
MQFB_PAN	275	X'00000113'
MQFB_NAN	276	X'00000114'
MQFB_DATA_LENGTH_ZERO	291	X'00000123'
MQFB_DATA_LENGTH_NEGATIVE	292	X'00000124'
MQFB_DATA_LENGTH_TOO_BIG	293	X'00000125'
MQFB_BUFFER_OVERFLOW	294	X'00000126'
MQFB_LENGTH_OFF_BY_ONE	295	X'00000127'
MQFB_IIH_ERROR	296	X'00000128'
MQFB_NOT_AUTHORIZED_FOR_IMS	298	X'0000012A'
MQFB_IMS_ERROR	300	X'0000012C'
MQFB_IMS_FIRST	301	X'0000012D'
MQFB_IMS_LAST	399	X'0000018F'
MQFB_CICS_INTERNAL_ERROR	401	X'00000191'
MQFB_CICS_NOT_AUTHORIZED	402	X'00000192'
MQFB_CICS_BRIDGE_FAILURE	403	X'00000193'
MQFB_CICS_CORREL_ID_ERROR	404	X'00000194'
MQFB_CICS_CCSID_ERROR	405	X'00000195'
MQFB_CICS_ENCODING_ERROR	406	X'00000196'
MQFB_CICS_CIH_ERROR	407	X'00000197'
MQFB_CICS_UOW_ERROR	408	X'00000198'
MQFB_CICS_COMMAREA_ERROR	409	X'00000199'
MQFB_CICS_APPL_NOT_STARTED	410	X'0000019A'
MQFB_CICS_APPL_ABENDED	411	X'0000019B'
MQFB_CICS_DLQ_ERROR	412	X'0000019C'
MQFB_CICS_UOW_BACKED_OUT	413	X'0000019D'
MQFB_SYSTEM_LAST	65535	X'0000FFFF'
MQFB_APPL_FIRST	65536	X'00010000'
MQFB_APPL_LAST	999999999	X'3B9AC9FF'

MQFMT_★ (Format)

See the *Format* field described in "MQMD – Message descriptor" on page 98.

MQFMT_NONE	'␣␣␣␣␣␣␣␣'
MQFMT_ADMIN	'MQADMIN␣'
MQFMT_CHANNEL_COMPLETED	'MQCHCOM␣'
MQFMT_CICS	'MQCICS␣␣'
MQFMT_COMMAND_1	'MQCMD1␣␣'
MQFMT_COMMAND_2	'MQCMD2␣␣'
MQFMT_DEAD_LETTER_HEADER	'MQDEAD␣␣'
MQFMT_DIST_HEADER	'MQHDIST␣'
MQFMT_EVENT	'MQEVENT␣'
MQFMT_IMS	'MQIMS␣␣␣'
MQFMT_IMS_VAR_STRING	'MQIMSVS␣'
MQFMT_MD_EXTENSION	'MQHMDE␣␣'
MQFMT_PCF	'MQPCF␣␣␣'
MQFMT_REF_MSG_HEADER	'MQHREF␣␣'
MQFMT_STRING	'MQSTR␣␣␣'
MQFMT_TRIGGER	'MQTRIG␣␣'
MQFMT_XMIT_Q_HEADER	'MQXMIT␣␣'

For the C programming language, the following are also defined:

MQFMT_NONE_ARRAY	'␣','␣','␣','␣','␣','␣','␣','␣'
MQFMT_ADMIN_ARRAY	'M','Q','A','D','M','I','N','␣'
MQFMT_CHANNEL_COMPLETED_ARRAY	'M','Q','C','H','C','O','M','␣'
MQFMT_CICS_ARRAY	'M','Q','C','I','C','S','␣','␣'
MQFMT_COMMAND_1_ARRAY	'M','Q','C','M','D','1','␣','␣'
MQFMT_COMMAND_2_ARRAY	'M','Q','C','M','D','2','␣','␣'
MQFMT_DEAD_LETTER_HEADER_ARRAY	'M','Q','D','E','A','D','␣','␣'
MQFMT_DIST_HEADER_ARRAY	'M','Q','H','D','I','S','T','␣'
MQFMT_EVENT_ARRAY	'M','Q','E','V','E','N','T','␣'
MQFMT_IMS_ARRAY	'M','Q','I','M','S','␣','␣','␣'
MQFMT_IMS_VAR_STRING_ARRAY	'M','Q','I','M','S','V','S','␣'
MQFMT_MD_EXTENSION_ARRAY	'M','Q','H','M','D','E','␣','␣'
MQFMT_PCF_ARRAY	'M','Q','P','C','F','␣','␣','␣'
MQFMT_REF_MSG_HEADER_ARRAY	'M','Q','H','R','E','F','␣','␣'
MQFMT_STRING_ARRAY	'M','Q','S','T','R','␣','␣','␣'
MQFMT_TRIGGER_ARRAY	'M','Q','T','R','I','G','␣','␣'
MQFMT_XMIT_Q_HEADER_ARRAY	'M','Q','X','M','I','T','␣','␣'

MQGI_★ (Group identifier)

See the *GroupId* field described in "MQMD – Message descriptor" on page 98.

MQGI_NONE	X'00...00' (24 nulls)

For the C programming language, the following is also defined:

MQGI_NONE_ARRAY	'\0','\0',...'\0','\0'

MQGMO_★ (Get message options)

See the *Options* field described in "MQGMO – Get-message options" on page 56.

MQGMO_NO_WAIT	0	X'00000000'
MQGMO_NONE	0	X'00000000'
MQGMO_WAIT	1	X'00000001'
MQGMO_SYNCPOINT	2	X'00000002'
MQGMO_NO_SYNCPOINT	4	X'00000004'
MQGMO_SET_SIGNAL	8	X'00000008'
MQGMO_BROWSE_FIRST	16	X'00000010'
MQGMO_BROWSE_NEXT	32	X'00000020'
MQGMO_ACCEPT_TRUNCATED_MSG	64	X'00000040'
MQGMO_MARK_SKIP_BACKOUT	128	X'00000080'
MQGMO_MSG_UNDER_CURSOR	256	X'00000100'
MQGMO_LOCK	512	X'00000200'
MQGMO_UNLOCK	1024	X'00000400'
MQGMO_BROWSE_MSG_UNDER_CURSOR	2048	X'00000800'
MQGMO_SYNCPOINT_IF_PERSISTENT	4096	X'00001000'
MQGMO_FAIL_IF_QUIESCING	8192	X'00002000'
MQGMO_CONVERT	16384	X'00004000'
MQGMO_LOGICAL_ORDER	32768	X'00008000'
MQGMO_COMPLETE_MSG	65536	X'00010000'
MQGMO_ALL_MSGS_AVAILABLE	131072	X'00020000'
MQGMO_ALL_SEGMENTS_AVAILABLE	262144	X'00040000'

MQGMO_★ (Get message options structure identifier)

See the *StrucId* field described in "MQGMO – Get-message options" on page 56.

MQGMO_STRUC_ID	'GMOb'

For the C programming language, the following is also defined:

MQGMO_STRUC_ID_ARRAY	'G','M','O','b'

MQGMO_★ (Get message options version)

See the *Version* field described in "MQGMO – Get-message options" on page 56.

MQGMO_VERSION_1	1	X'00000001'
MQGMO_VERSION_2	2	X'00000002'
MQGMO_CURRENT_VERSION	2	X'00000002'

MQGS_★ (Group status)

See the *GroupStatus* field described in "MQGMO – Get-message options" on page 56.

MQGS_NOT_IN_GROUP	'b'
MQGS_MSG_IN_GROUP	'G'
MQGS_LAST_MSG_IN_GROUP	'L'

MQHC_★ (Connection handle)

See the *Hconn* parameter described in "MQCONN – Connect queue manager" on page 261 and "MQDISC – Disconnect queue manager" on page 269.

MQHC_UNUSABLE_HCONN	-1	X'FFFFFFFF'
MQHC_DEF_HCONN	0	X'00000000'

MQHO_★ (Object handle)

See the *Hobj* parameter described in "MQCLOSE – Close object" on page 248.

MQHO_UNUSABLE_HOBJ	-1	X'FFFFFFFF'

MQIA_★ (Integer attribute selector)

See the *Selectors* parameter described in "MQINQ – Inquire about object attributes" on page 285 and "MQSET – Set object attributes" on page 333.

MQIA_FIRST	1	X'00000001'
MQIA_APPL_TYPE	1	X'00000001'
MQIA_CODED_CHAR_SET_ID	2	X'00000002'
MQIA_CURRENT_Q_DEPTH	3	X'00000003'
MQIA_DEF_INPUT_OPEN_OPTION	4	X'00000004'
MQIA_DEF_PERSISTENCE	5	X'00000005'
MQIA_DEF_PRIORITY	6	X'00000006'
MQIA_DEFINITION_TYPE	7	X'00000007'
MQIA_HARDEN_GET_BACKOUT	8	X'00000008'
MQIA_INHIBIT_GET	9	X'00000009'
MQIA_INHIBIT_PUT	10	X'0000000A'
MQIA_MAX_HANDLES	11	X'0000000B'
MQIA_USAGE	12	X'0000000C'
MQIA_MAX_MSG_LENGTH	13	X'0000000D'
MQIA_MAX_PRIORITY	14	X'0000000E'
MQIA_MAX_Q_DEPTH	15	X'0000000F'
MQIA_MSG_DELIVERY_SEQUENCE	16	X'00000010'
MQIA_OPEN_INPUT_COUNT	17	X'00000011'
MQIA_OPEN_OUTPUT_COUNT	18	X'00000012'
MQIA_NAME_COUNT	19	X'00000013'
MQIA_Q_TYPE	20	X'00000014'
MQIA_RETENTION_INTERVAL	21	X'00000015'
MQIA_BACKOUT_THRESHOLD	22	X'00000016'
MQIA_SHAREABILITY	23	X'00000017'
MQIA_TRIGGER_CONTROL	24	X'00000018'
MQIA_TRIGGER_INTERVAL	25	X'00000019'
MQIA_TRIGGER_MSG_PRIORITY	26	X'0000001A'
MQIA_TRIGGER_TYPE	28	X'0000001C'
MQIA_TRIGGER_DEPTH	29	X'0000001D'
MQIA_SYNCPOINT	30	X'0000001E'
MQIA_COMMAND_LEVEL	31	X'0000001F'
MQIA_PLATFORM	32	X'00000020'
MQIA_MAX_UNCOMMITTED_MSGS	33	X'00000021'
MQIA_DIST_LISTS	34	X'00000022'
MQIA_TIME_SINCE_RESET	35	X'00000023'
MQIA_HIGH_Q_DEPTH	36	X'00000024'

MQIA_MSG_ENQ_COUNT	37	X'00000025'
MQIA_MSG_DEQ_COUNT	38	X'00000026'
MQIA_Q_DEPTH_HIGH_LIMIT	40	X'00000028'
MQIA_Q_DEPTH_LOW_LIMIT	41	X'00000029'
MQIA_Q_DEPTH_MAX_EVENT	42	X'0000002A'
MQIA_Q_DEPTH_HIGH_EVENT	43	X'0000002B'
MQIA_Q_DEPTH_LOW_EVENT	44	X'0000002C'
MQIA_SCOPE	45	X'0000002D'
MQIA_Q_SERVICE_INTERVAL_EVENT	46	X'0000002E'
MQIA_AUTHORITY_EVENT	47	X'0000002F'
MQIA_INHIBIT_EVENT	48	X'00000030'
MQIA_LOCAL_EVENT	49	X'00000031'
MQIA_REMOTE_EVENT	50	X'00000032'
MQIA_START_STOP_EVENT	52	X'00000034'
MQIA_PERFORMANCE_EVENT	53	X'00000035'
MQIA_Q_SERVICE_INTERVAL	54	X'00000036'
MQIA_CHANNEL_AUTO_DEF	55	X'00000037'
MQIA_CHANNEL_AUTO_DEF_EVENT	56	X'00000038'
MQIA_INDEX_TYPE	57	X'00000039'
MQIA_LAST	2000	X'000007D0'
MQIA_LAST_USED	(environment specific)	

MQIAUT_★ (IMS authenticator)

See the *Authenticator* field described in "MQIIH – IMS bridge header" on page 91.

MQIAUT_NONE	'␣␣␣␣␣␣␣␣'

For the C programming language, the following is also defined:

MQIAUT_NONE_ARRAY	'␣','␣','␣','␣','␣','␣','␣','␣'

MQIAV_★ (Integer attribute value)

See the *IntAttrs* parameter described in "MQINQ – Inquire about object attributes" on page 285.

MQIAV_UNDEFINED	-2	X'FFFFFFFE'
MQIAV_NOT_APPLICABLE	-1	X'FFFFFFFF'

MQICM_★ (IMS commit mode)

See the *CommitMode* field described in "MQIIH – IMS bridge header" on page 91.

MQICM_COMMIT_THEN_SEND	'0'
MQICM_SEND_THEN_COMMIT	'1'

MQIIH_★ (IMS header flags)

See the *Flags* field described in "MQIIH – IMS bridge header" on page 91.

MQIIH_NONE	0	X'00000000'

MQIIH_★ (IMS header length)

See the *StrucLength* field described in "MQIIH – IMS bridge header" on page 91.

MQIIH_LENGTH_1	84	X'00000054'

MQIIH_★ (IMS header structure identifier)

See the *StrucId* field described in "MQIIH – IMS bridge header" on page 91.

MQIIH_STRUC_ID	'IIHb'

For the C programming language, the following is also defined:

MQIIH_STRUC_ID_ARRAY	'I','I','H','b'

MQIIH_★ (IMS header version)

See the *Version* field described in "MQIIH – IMS bridge header" on page 91.

MQIIH_VERSION_1	1	X'00000001'
MQIIH_CURRENT_VERSION	1	X'00000001'

MQISS_★ (IMS security scope)

See the *SecurityScope* field described in "MQIIH – IMS bridge header" on page 91.

MQISS_CHECK	'C'
MQISS_FULL	'F'

MQIT_★ (Index type)

See the *IndexType* attribute described in "Attributes for local queues and model queues" on page 348.

MQIT_NONE	0
MQIT_MSG_ID	1
MQIT_CORREL_ID	2

MQITII_★ (IMS transaction instance identifier)

See the *TranInstanceId* field described in "MQIIH – IMS bridge header" on page 91.

MQITII_NONE	X'00...00' (16 nulls)

For the C programming language, the following is also defined:

MQITII_NONE_ARRAY	'\0','\0',...'\0','\0'

MQITS_★ (IMS transaction state)

See the *TranState* field described in "MQIIH – IMS bridge header" on page 91.

MQITS_IN_CONVERSATION	'C'
MQITS_NOT_IN_CONVERSATION	' '

MQMD_★ (Message descriptor structure identifier)

See the *StrucId* field described in "MQMD – Message descriptor" on page 98.

MQMD_STRUC_ID	'MDbb'

For the C programming language, the following is also defined:

MQMD_STRUC_ID_ARRAY	'M','D','b','b'

MQMD_★ (Message descriptor version)

See the *Version* field described in "MQMD – Message descriptor" on page 98.

MQMD_VERSION_1	1	X'00000001'
MQMD_VERSION_2	2	X'00000002'
MQMD_CURRENT_VERSION	2	X'00000002'

MQMDE_★ (Message descriptor extension length)

See the *StrucLength* field described in "MQMDE – Message descriptor extension" on page 153.

MQMDE_LENGTH_2	72	X'00000048'

MQMDE_★ (Message descriptor extension structure identifier)

See the *StrucId* field described in "MQMDE – Message descriptor extension" on page 153.

MQMDE_STRUC_ID	'MDEb'

For the C programming language, the following is also defined:

MQMDE_STRUC_ID_ARRAY	'M','D','E','b'

MQMDE_★ (Message descriptor extension version)

See the *Version* field described in "MQMDE – Message descriptor extension" on page 153.

MQMDE_VERSION_2	2	X'00000002'
MQMDE_CURRENT_VERSION	2	X'00000002'

MQMDEF_★ (Message descriptor extension flags)

See the *Flags* field described in "MQMDE – Message descriptor extension" on page 153.

MQMDEF_NONE	0	X'00000000'

MQMDS_★ (Message delivery sequence)

See the *MsgDeliverySequence* attribute described in "Attributes for local queues and model queues" on page 348.

MQMDS_PRIORITY	0	X'00000000'
MQMDS_FIFO	1	X'00000001'

MQMF_★ (Message flags)

See the *MsgFlags* field described in "MQMD – Message descriptor" on page 98.

MQMF_SEGMENTATION_INHIBITED	0	X'00000000'
MQMF_NONE	0	X'00000000'
MQMF_SEGMENTATION_ALLOWED	1	X'00000001'
MQMF_SEGMENT	2	X'00000002'
MQMF_LAST_SEGMENT	4	X'00000004'
MQMF_MSG_IN_GROUP	8	X'00000008'
MQMF_LAST_MSG_IN_GROUP	16	X'00000010'

MQMF_★ (Message-flags masks)

See Appendix C, "Report options and message flags" on page 489.

MQMF_ACCEPT_UNSUP_MASK	-1048576	X'FFF00000'
MQMF_ACCEPT_UNSUP_IF_XMIT_MASK	1044480	X'000FF000'
MQMF_REJECT_UNSUP_MASK	4095	X'00000FFF'

MQMI_★ (Message identifier)

See the *MsgId* field described in "MQMD – Message descriptor" on page 98.

MQMI_NONE	X'00...00' (24 nulls)

For the C programming language, the following is also defined:

MQMI_NONE_ARRAY	'\0','\0',...'\0','\0'

MQMO_★ (Match options)

See the *MatchOptions* field described in "MQGMO – Get-message options" on page 56.

MQMO_NONE	0	X'00000000'
MQMO_MATCH_MSG_ID	1	X'00000001'
MQMO_MATCH_CORREL_ID	2	X'00000002'
MQMO_MATCH_GROUP_ID	4	X'00000004'
MQMO_MATCH_MSG_SEQ_NUMBER	8	X'00000008'
MQMO_MATCH_OFFSET	16	X'00000010'

MQMT_★ (Message type)

See the *MsgType* field described in "MQMD – Message descriptor" on page 98.

MQMT_SYSTEM_FIRST	1	X'00000001'
MQMT_REQUEST	1	X'00000001'
MQMT_REPLY	2	X'00000002'
MQMT_REPORT	4	X'00000004'
MQMT_DATAGRAM	8	X'00000008'
MQMT_SYSTEM_LAST	65535	X'0000FFFF'
MQMT_APPL_FIRST	65536	X'00010000'
MQMT_APPL_LAST	999999999	X'3B9AC9FF'

MQOD_★ (Object descriptor length)

MQOD_CURRENT_LENGTH	(environment specific)

MQOD_★ (Object descriptor structure identifier)

See the *StrucId* field described in "MQOD – Object descriptor" on page 160.

MQOD_STRUC_ID	'ODbb'

For the C programming language, the following is also defined:

MQOD_STRUC_ID_ARRAY	'O','D','b','b'

MQOD_★ (Object descriptor version)

See the *Version* field described in "MQOD – Object descriptor" on page 160.

MQOD_VERSION_1	1	X'00000001'
MQOD_VERSION_2	2	X'00000002'
MQOD_CURRENT_VERSION	2	X'00000002'

MQOII_★ (Object instance identifier)

See the *ObjectInstanceId* field described in "MQRMH – Message reference header" on page 197.

MQOII_NONE	X'00...00' (24 nulls)

For the C programming language, the following is also defined:

MQOII_NONE_ARRAY	'\0','\0',...'\0','\0'

MQOL_★ (Original length)

See the *OriginalLength* field described in "MQMD – Message descriptor" on page 98.

MQOL_UNDEFINED	-1	X'FFFFFFFF'

MQOO_★ (Open options)

See the *Options* parameter described in "MQOPEN – Open object" on page 297.

MQOO_INPUT_AS_Q_DEF	1	X'00000001'
MQOO_INPUT_SHARED	2	X'00000002'
MQOO_INPUT_EXCLUSIVE	4	X'00000004'
MQOO_BROWSE	8	X'00000008'
MQOO_OUTPUT	16	X'00000010'
MQOO_INQUIRE	32	X'00000020'
MQOO_SET	64	X'00000040'
MQOO_SAVE_ALL_CONTEXT	128	X'00000080'
MQOO_PASS_IDENTITY_CONTEXT	256	X'00000100'
MQOO_PASS_ALL_CONTEXT	512	X'00000200'
MQOO_SET_IDENTITY_CONTEXT	1024	X'00000400'
MQOO_SET_ALL_CONTEXT	2048	X'00000800'
MQOO_ALTERNATE_USER_AUTHORITY	4096	X'00001000'

MQOO_FAIL_IF_QUIESCING	8192	X'00002000'

MQOT_★ (Object type)

See the *ObjectType* field described in "MQOD – Object descriptor" on page 160.

MQOT_Q	1	X'00000001'
MQOT_NAMELIST	2	X'00000002'
MQOT_PROCESS	3	X'00000003'
MQOT_Q_MGR	5	X'00000005'
MQOT_CHANNEL	6	X'00000006'
MQOT_RESERVED_1	7	X'00000007'

MQPER_★ (Persistence)

See the *Persistence* field described in "MQMD – Message descriptor" on page 98, and the *DefPersistence* attribute described in "Attributes for all queues" on page 343.

MQPER_NOT_PERSISTENT	0	X'00000000'
MQPER_PERSISTENT	1	X'00000001'
MQPER_PERSISTENCE_AS_Q_DEF	2	X'00000002'

MQPL_★ (Platform)

See the *Platform* attribute described in "Attributes for the queue manager" on page 370.

MQPL_MVS	1	X'00000001'
MQPL_OS2	2	X'00000002'
MQPL_AIX	3	X'00000003'
MQPL_UNIX	3	X'00000003'
MQPL_OS400	4	X'00000004'
MQPL_WINDOWS	5	X'00000005'
MQPL_WINDOWS_NT	11	X'0000000B'
MQPL_VMS	12	X'0000000C'
MQPL_NSK	13	X'0000000D'

MQPMO_★ (Put message options)

See the *Options* field described in "MQPMO – Put message options" on page 173.

MQPMO_NONE	0	X'00000000'
MQPMO_SYNCPOINT	2	X'00000002'
MQPMO_NO_SYNCPOINT	4	X'00000004'
MQPMO_DEFAULT_CONTEXT	32	X'00000020'
MQPMO_NEW_MSG_ID	64	X'00000040'
MQPMO_NEW_CORREL_ID	128	X'00000080'
MQPMO_PASS_IDENTITY_CONTEXT	256	X'00000100'
MQPMO_PASS_ALL_CONTEXT	512	X'00000200'
MQPMO_SET_IDENTITY_CONTEXT	1024	X'00000400'
MQPMO_SET_ALL_CONTEXT	2048	X'00000800'
MQPMO_ALTERNATE_USER_AUTHORITY	4096	X'00001000'
MQPMO_FAIL_IF_QUIESCING	8192	X'00002000'
MQPMO_NO_CONTEXT	16384	X'00004000'
MQPMO_LOGICAL_ORDER	32768	X'00008000'

MQPMO_★ (Put message options structure length)

MQPMO_CURRENT_LENGTH	(environment specific)

MQPMO_★ (Put message options structure identifier)

See the *StrucId* field described in "MQPMO – Put message options" on page 173.

MQPMO_STRUC_ID	'PMOb'

For the C programming language, the following is also defined:

MQPMO_STRUC_ID_ARRAY	'P','M','O','b'

MQPMO_★ (Put message options version)

See the *Version* field described in "MQPMO – Put message options" on page 173.

MQPMO_VERSION_1	1	X'00000001'
MQPMO_VERSION_2	2	X'00000002'
MQPMO_CURRENT_VERSION	2	X'00000002'

MQPMRF_★ (Put message record field flags)

See the *PutMsgRecFields* field described in "MQDH – Distribution header" on page 39.

MQPMRF_NONE	0	X'00000000'
MQPMRF_MSG_ID	1	X'00000001'
MQPMRF_CORREL_ID	2	X'00000002'
MQPMRF_GROUP_ID	4	X'00000004'
MQPMRF_FEEDBACK	8	X'00000008'
MQPMRF_ACCOUNTING_TOKEN	16	X'00000010'

MQPRI_★ (Priority)

See the *Priority* field described in "MQMD – Message descriptor" on page 98.

MQPRI_PRIORITY_AS_Q_DEF	-1	X'FFFFFFFF'

MQQA_★ (Inhibit get)

See the *InhibitGet* attribute described in "Attributes for all queues" on page 343.

MQQA_GET_ALLOWED	0	X'00000000'
MQQA_GET_INHIBITED	1	X'00000001'

MQQA_★ (Inhibit put)

See the *InhibitPut* attribute described in "Attributes for all queues" on page 343.

MQQA_PUT_ALLOWED	0	X'00000000'
MQQA_PUT_INHIBITED	1	X'00000001'

MQQA_★ (Backout hardening)

See the *HardenGetBackout* attribute described in "Attributes for local queues and model queues" on page 348.

MQQA_BACKOUT_NOT_HARDENED	0	X'00000000'
MQQA_BACKOUT_HARDENED	1	X'00000001'

MQQA_★ (Queue shareability)

See the *Shareability* attribute described in "Attributes for local queues and model queues" on page 348.

MQQA_NOT_SHAREABLE	0	X'00000000'
MQQA_SHAREABLE	1	X'00000001'

MQQDT_★ (Queue definition type)

See the *DefinitionType* attribute described in "Attributes for local queues and model queues" on page 348.

MQQDT_PREDEFINED	1	X'00000001'
MQQDT_PERMANENT_DYNAMIC	2	X'00000002'
MQQDT_TEMPORARY_DYNAMIC	3	X'00000003'

MQQSIE_★ (Service interval events)

MQQSIE_NONE	0	X'00000000'
MQQSIE_HIGH	1	X'00000001'
MQQSIE_OK	2	X'00000002'

MQQT_★ (Queue type)

See the *QType* attribute described in "Attributes for all queues" on page 343.

MQQT_LOCAL	1	X'00000001'
MQQT_MODEL	2	X'00000002'
MQQT_ALIAS	3	X'00000003'
MQQT_REMOTE	6	X'00000006'

MQRC_★ (Reason code)

See Chapter 5, "Return codes" on page 383, and the *Feedback* field described in "MQMD – Message descriptor" on page 98. Note: the following list is in **numeric order**.

MQRC_NONE	0	X'00000000'
MQRC_ALIAS_BASE_Q_TYPE_ERROR	2001	X'000007D1'
MQRC_ALREADY_CONNECTED	2002	X'000007D2'
MQRC_BACKED_OUT	2003	X'000007D3'
MQRC_BUFFER_ERROR	2004	X'000007D4'
MQRC_BUFFER_LENGTH_ERROR	2005	X'000007D5'
MQRC_CHAR_ATTR_LENGTH_ERROR	2006	X'000007D6'
MQRC_CHAR_ATTRS_ERROR	2007	X'000007D7'
MQRC_CHAR_ATTRS_TOO_SHORT	2008	X'000007D8'
MQRC_CONNECTION_BROKEN	2009	X'000007D9'
MQRC_DATA_LENGTH_ERROR	2010	X'000007DA'

MQRC_DYNAMIC_Q_NAME_ERROR	2011	X'000007DB'
MQRC_ENVIRONMENT_ERROR	2012	X'000007DC'
MQRC_EXPIRY_ERROR	2013	X'000007DD'
MQRC_FEEDBACK_ERROR	2014	X'000007DE'
MQRC_GET_INHIBITED	2016	X'000007E0'
MQRC_HANDLE_NOT_AVAILABLE	2017	X'000007E1'
MQRC_HCONN_ERROR	2018	X'000007E2'
MQRC_HOBJ_ERROR	2019	X'000007E3'
MQRC_INHIBIT_VALUE_ERROR	2020	X'000007E4'
MQRC_INT_ATTR_COUNT_ERROR	2021	X'000007E5'
MQRC_INT_ATTR_COUNT_TOO_SMALL	2022	X'000007E6'
MQRC_INT_ATTRS_ARRAY_ERROR	2023	X'000007E7'
MQRC_SYNCPOINT_LIMIT_REACHED	2024	X'000007E8'
MQRC_MAX_CONNS_LIMIT_REACHED	2025	X'000007E9'
MQRC_MD_ERROR	2026	X'000007EA'
MQRC_MISSING_REPLY_TO_Q	2027	X'000007EB'
MQRC_MSG_TYPE_ERROR	2029	X'000007ED'
MQRC_MSG_TOO_BIG_FOR_Q	2030	X'000007EE'
MQRC_MSG_TOO_BIG_FOR_Q_MGR	2031	X'000007EF'
MQRC_NO_MSG_AVAILABLE	2033	X'000007F1'
MQRC_NO_MSG_UNDER_CURSOR	2034	X'000007F2'
MQRC_NOT_AUTHORIZED	2035	X'000007F3'
MQRC_NOT_OPEN_FOR_BROWSE	2036	X'000007F4'
MQRC_NOT_OPEN_FOR_INPUT	2037	X'000007F5'
MQRC_NOT_OPEN_FOR_INQUIRE	2038	X'000007F6'
MQRC_NOT_OPEN_FOR_OUTPUT	2039	X'000007F7'
MQRC_NOT_OPEN_FOR_SET	2040	X'000007F8'
MQRC_OBJECT_CHANGED	2041	X'000007F9'
MQRC_OBJECT_IN_USE	2042	X'000007FA'
MQRC_OBJECT_TYPE_ERROR	2043	X'000007FB'
MQRC_OD_ERROR	2044	X'000007FC'
MQRC_OPTION_NOT_VALID_FOR_TYPE	2045	X'000007FD'
MQRC_OPTIONS_ERROR	2046	X'000007FE'
MQRC_PERSISTENCE_ERROR	2047	X'000007FF'
MQRC_PERSISTENT_NOT_ALLOWED	2048	X'00000800'
MQRC_PRIORITY_EXCEEDS_MAXIMUM	2049	X'00000801'
MQRC_PRIORITY_ERROR	2050	X'00000802'
MQRC_PUT_INHIBITED	2051	X'00000803'
MQRC_Q_DELETED	2052	X'00000804'
MQRC_Q_FULL	2053	X'00000805'
MQRC_Q_NOT_EMPTY	2055	X'00000807'
MQRC_Q_SPACE_NOT_AVAILABLE	2056	X'00000808'
MQRC_Q_TYPE_ERROR	2057	X'00000809'
MQRC_Q_MGR_NAME_ERROR	2058	X'0000080A'
MQRC_Q_MGR_NOT_AVAILABLE	2059	X'0000080B'
MQRC_REPORT_OPTIONS_ERROR	2061	X'0000080D'
MQRC_SECOND_MARK_NOT_ALLOWED	2062	X'0000080E'
MQRC_SECURITY_ERROR	2063	X'0000080F'
MQRC_SELECTOR_COUNT_ERROR	2065	X'00000811'
MQRC_SELECTOR_LIMIT_EXCEEDED	2066	X'00000812'
MQRC_SELECTOR_ERROR	2067	X'00000813'
MQRC_SELECTOR_NOT_FOR_TYPE	2068	X'00000814'
MQRC_SIGNAL_OUTSTANDING	2069	X'00000815'
MQRC_SIGNAL_REQUEST_ACCEPTED	2070	X'00000816'

MQRC_STORAGE_NOT_AVAILABLE	2071	X'00000817'
MQRC_SYNCPOINT_NOT_AVAILABLE	2072	X'00000818'
MQRC_TRIGGER_CONTROL_ERROR	2075	X'0000081B'
MQRC_TRIGGER_DEPTH_ERROR	2076	X'0000081C'
MQRC_TRIGGER_MSG_PRIORITY_ERR	2077	X'0000081D'
MQRC_TRIGGER_TYPE_ERROR	2078	X'0000081E'
MQRC_TRUNCATED_MSG_ACCEPTED	2079	X'0000081F'
MQRC_TRUNCATED_MSG_FAILED	2080	X'00000820'
MQRC_UNKNOWN_ALIAS_BASE_Q	2082	X'00000822'
MQRC_UNKNOWN_OBJECT_NAME	2085	X'00000825'
MQRC_UNKNOWN_OBJECT_Q_MGR	2086	X'00000826'
MQRC_UNKNOWN_REMOTE_Q_MGR	2087	X'00000827'
MQRC_WAIT_INTERVAL_ERROR	2090	X'0000082A'
MQRC_XMIT_Q_TYPE_ERROR	2091	X'0000082B'
MQRC_XMIT_Q_USAGE_ERROR	2092	X'0000082C'
MQRC_NOT_OPEN_FOR_PASS_ALL	2093	X'0000082D'
MQRC_NOT_OPEN_FOR_PASS_IDENT	2094	X'0000082E'
MQRC_NOT_OPEN_FOR_SET_ALL	2095	X'0000082F'
MQRC_NOT_OPEN_FOR_SET_IDENT	2096	X'00000830'
MQRC_CONTEXT_HANDLE_ERROR	2097	X'00000831'
MQRC_CONTEXT_NOT_AVAILABLE	2098	X'00000832'
MQRC_SIGNAL1_ERROR	2099	X'00000833'
MQRC_OBJECT_ALREADY_EXISTS	2100	X'00000834'
MQRC_OBJECT_DAMAGED	2101	X'00000835'
MQRC_RESOURCE_PROBLEM	2102	X'00000836'
MQRC_ANOTHER_Q_MGR_CONNECTED	2103	X'00000837'
MQRC_UNKNOWN_REPORT_OPTION	2104	X'00000838'
MQRC_STORAGE_CLASS_ERROR	2105	X'00000839'
MQRC_COD_NOT_VALID_FOR_XCF_Q	2106	X'0000083A'
MQRC_XWAIT_CANCELED	2107	X'0000083B'
MQRC_XWAIT_ERROR	2108	X'0000083C'
MQRC_SUPPRESSED_BY_EXIT	2109	X'0000083D'
MQRC_FORMAT_ERROR	2110	X'0000083E'
MQRC_SOURCE_CCSID_ERROR	2111	X'0000083F'
MQRC_SOURCE_INTEGER_ENC_ERROR	2112	X'00000840'
MQRC_SOURCE_DECIMAL_ENC_ERROR	2113	X'00000841'
MQRC_SOURCE_FLOAT_ENC_ERROR	2114	X'00000842'
MQRC_TARGET_CCSID_ERROR	2115	X'00000843'
MQRC_TARGET_INTEGER_ENC_ERROR	2116	X'00000844'
MQRC_TARGET_DECIMAL_ENC_ERROR	2117	X'00000845'
MQRC_TARGET_FLOAT_ENC_ERROR	2118	X'00000846'
MQRC_NOT_CONVERTED	2119	X'00000847'
MQRC_CONVERTED_MSG_TOO_BIG	2120	X'00000848'
MQRC_TRUNCATED	2120	X'00000848'
MQRC_NO_EXTERNAL_PARTICIPANTS	2121	X'00000849'
MQRC_PARTICIPANT_NOT_AVAILABLE	2122	X'0000084A'
MQRC_OUTCOME_MIXED	2123	X'0000084B'
MQRC_OUTCOME_PENDING	2124	X'0000084C'
MQRC_BRIDGE_STARTED	2125	X'0000084D'
MQRC_BRIDGE_STOPPED	2126	X'0000084E'
MQRC_ADAPTER_STORAGE_SHORTAGE	2127	X'0000084F'
MQRC_UOW_IN_PROGRESS	2128	X'00000850'
MQRC_ADAPTER_CONN_LOAD_ERROR	2129	X'00000851'
MQRC_ADAPTER_SERV_LOAD_ERROR	2130	X'00000852'

MQRC_ADAPTER_DEFS_ERROR	2131	X'00000853'
MQRC_ADAPTER_DEFS_LOAD_ERROR	2132	X'00000854'
MQRC_ADAPTER_CONV_LOAD_ERROR	2133	X'00000855'
MQRC_BO_ERROR	2134	X'00000856'
MQRC_DH_ERROR	2135	X'00000857'
MQRC_MULTIPLE_REASONS	2136	X'00000858'
MQRC_OPEN_FAILED	2137	X'00000859'
MQRC_ADAPTER_DISC_LOAD_ERROR	2138	X'0000085A'
MQRC_CNO_ERROR	2139	X'0000085B'
MQRC_CICS_WAIT_FAILED	2140	X'0000085C'
MQRC_DLH_ERROR	2141	X'0000085D'
MQRC_HEADER_ERROR	2142	X'0000085E'
MQRC_SOURCE_LENGTH_ERROR	2143	X'0000085F'
MQRC_TARGET_LENGTH_ERROR	2144	X'00000860'
MQRC_SOURCE_BUFFER_ERROR	2145	X'00000861'
MQRC_TARGET_BUFFER_ERROR	2146	X'00000862'
MQRC_IIH_ERROR	2148	X'00000864'
MQRC_PCF_ERROR	2149	X'00000865'
MQRC_DBCS_ERROR	2150	X'00000866'
MQRC_OBJECT_NAME_ERROR	2152	X'00000868'
MQRC_OBJECT_Q_MGR_NAME_ERROR	2153	X'00000869'
MQRC_RECS_PRESENT_ERROR	2154	X'0000086A'
MQRC_OBJECT_RECORDS_ERROR	2155	X'0000086B'
MQRC_RESPONSE_RECORDS_ERROR	2156	X'0000086C'
MQRC_ASID_MISMATCH	2157	X'0000086D'
MQRC_PMO_RECORD_FLAGS_ERROR	2158	X'0000086E'
MQRC_PUT_MSG_RECORDS_ERROR	2159	X'0000086F'
MQRC_CONN_ID_IN_USE	2160	X'00000870'
MQRC_Q_MGR_QUIESCING	2161	X'00000871'
MQRC_Q_MGR_STOPPING	2162	X'00000872'
MQRC_DUPLICATE_RECOV_COORD	2163	X'00000873'
MQRC_PMO_ERROR	2173	X'0000087D'
MQRC_API_EXIT_LOAD_ERROR	2183	X'00000887'
MQRC_REMOTE_Q_NAME_ERROR	2184	X'00000888'
MQRC_INCONSISTENT_PERSISTENCE	2185	X'00000889'
MQRC_GMO_ERROR	2186	X'0000088A'
MQRC_CICS_BRIDGE_RESTRICTION	2187	X'0000088B'
MQRC_TMC_ERROR	2191	X'0000088F'
MQRC_PAGESET_FULL	2192	X'00000890'
MQRC_PAGESET_ERROR	2193	X'00000891'
MQRC_NAME_NOT_VALID_FOR_TYPE	2194	X'00000892'
MQRC_UNEXPECTED_ERROR	2195	X'00000893'
MQRC_UNKNOWN_XMIT_Q	2196	X'00000894'
MQRC_UNKNOWN_DEF_XMIT_Q	2197	X'00000895'
MQRC_DEF_XMIT_Q_TYPE_ERROR	2198	X'00000896'
MQRC_DEF_XMIT_Q_USAGE_ERROR	2199	X'00000897'
MQRC_NAME_IN_USE	2201	X'00000899'
MQRC_CONNECTION_QUIESCING	2202	X'0000089A'
MQRC_CONNECTION_STOPPING	2203	X'0000089B'
MQRC_ADAPTER_NOT_AVAILABLE	2204	X'0000089C'
MQRC_MSG_ID_ERROR	2206	X'0000089E'
MQRC_CORREL_ID_ERROR	2207	X'0000089F'
MQRC_FILE_SYSTEM_ERROR	2208	X'000008A0'
MQRC_NO_MSG_LOCKED	2209	X'000008A1'

MQRC_FILE_NOT_AUDITED	2216	X'000008A8'
MQRC_CONNECTION_NOT_AUTHORIZED	2217	X'000008A9'
MQRC_MSG_TOO_BIG_FOR_CHANNEL	2218	X'000008AA'
MQRC_CALL_IN_PROGRESS	2219	X'000008AB'
MQRC_RMH_ERROR	2220	X'000008AC'
MQRC_Q_MGR_ACTIVE	2222	X'000008AE'
MQRC_Q_MGR_NOT_ACTIVE	2223	X'000008AF'
MQRC_Q_DEPTH_HIGH	2224	X'000008B0'
MQRC_Q_DEPTH_LOW	2225	X'000008B1'
MQRC_Q_SERVICE_INTERVAL_HIGH	2226	X'000008B2'
MQRC_Q_SERVICE_INTERVAL_OK	2227	X'000008B3'
MQRC_UNIT_OF_WORK_NOT_STARTED	2232	X'000008B8'
MQRC_CHANNEL_AUTO_DEF_OK	2233	X'000008B9'
MQRC_CHANNEL_AUTO_DEF_ERROR	2234	X'000008BA'
MQRC_CFH_ERROR	2235	X'000008BB'
MQRC_CFIL_ERROR	2236	X'000008BC'
MQRC_CFIN_ERROR	2237	X'000008BD'
MQRC_CFSL_ERROR	2238	X'000008BE'
MQRC_CFST_ERROR	2239	X'000008BF'
MQRC_INCOMPLETE_GROUP	2241	X'000008C1'
MQRC_INCOMPLETE_MSG	2242	X'000008C2'
MQRC_INCONSISTENT_CCSIDS	2243	X'000008C3'
MQRC_INCONSISTENT_ENCODINGS	2244	X'000008C4'
MQRC_INCONSISTENT_UOW	2245	X'000008C5'
MQRC_INVALID_MSG_UNDER_CURSOR	2246	X'000008C6'
MQRC_MATCH_OPTIONS_ERROR	2247	X'000008C7'
MQRC_MDE_ERROR	2248	X'000008C8'
MQRC_MSG_FLAGS_ERROR	2249	X'000008C9'
MQRC_MSG_SEQ_NUMBER_ERROR	2250	X'000008CA'
MQRC_OFFSET_ERROR	2251	X'000008CB'
MQRC_ORIGINAL_LENGTH_ERROR	2252	X'000008CC'
MQRC_SEGMENT_LENGTH_ZERO	2253	X'000008CD'
MQRC_UOW_NOT_AVAILABLE	2255	X'000008CF'
MQRC_WRONG_GMO_VERSION	2256	X'000008D0'
MQRC_WRONG_MD_VERSION	2257	X'000008D1'
MQRC_GROUP_ID_ERROR	2258	X'000008D2'
MQRC_INCONSISTENT_BROWSE	2259	X'000008D3'
MQRC_XQH_ERROR	2260	X'000008D4'
MQRC_SRC_ENV_ERROR	2261	X'000008D5'
MQRC_SRC_NAME_ERROR	2262	X'000008D6'
MQRC_DEST_ENV_ERROR	2263	X'000008D7'
MQRC_DEST_NAME_ERROR	2264	X'000008D8'
MQRC_TM_ERROR	2265	X'000008D9'
MQRC_HCONFIG_ERROR	2280	X'000008E8'
MQRC_FUNCTION_ERROR	2281	X'000008E9'
MQRC_CHANNEL_STARTED	2282	X'000008EA'
MQRC_CHANNEL_STOPPED	2283	X'000008EB'
MQRC_CHANNEL_CONV_ERROR	2284	X'000008EC'
MQRC_SERVICE_NOT_AVAILABLE	2285	X'000008ED'
MQRC_INITIALIZATION_FAILED	2286	X'000008EE'
MQRC_TERMINATION_FAILED	2287	X'000008EF'
MQRC_UNKNOWN_Q_NAME	2288	X'000008F0'
MQRC_SERVICE_ERROR	2289	X'000008F1'
MQRC_Q_ALREADY_EXISTS	2290	X'000008F2'

MQRC_USER_ID_NOT_AVAILABLE	2291	X'000008F3'
MQRC_UNKNOWN_ENTITY	2292	X'000008F4'
MQRC_UNKNOWN_AUTH_ENTITY	2293	X'000008F5'
MQRC_UNKNOWN_REF_OBJECT	2294	X'000008F6'
MQRC_CHANNEL_ACTIVATED	2295	X'000008F7'
MQRC_CHANNEL_NOT_ACTIVATED	2296	X'000008F8'
MQRC_UOW_CANCELLED	2297	X'000008F9'

MQRMH_★ (Reference message header structure identifier)

See the *StrucId* field described in "MQRMH – Message reference header" on page 197.

MQRMH_STRUC_ID	'RMHb'

For the C programming language, the following is also defined:

MQRMH_STRUC_ID_ARRAY	'R','M','H','b'

MQRMH_★ (Reference message header version)

See the *Version* field described in "MQRMH – Message reference header" on page 197.

MQRMH_VERSION_1	1	X'00000001'
MQRMH_CURRENT_VERSION	1	X'00000001'

MQRMHF_★ (Reference message header flags)

See the *Flags* field described in "MQRMH – Message reference header" on page 197.

MQRMHF_NOT_LAST	0	X'00000000'
MQRMHF_LAST	1	X'00000001'

MQRO_★ (Report options)

See the *Report* field described in "MQMD – Message descriptor" on page 98.

MQRO_NEW_MSG_ID	0	X'00000000'
MQRO_COPY_MSG_ID_TO_CORREL_ID	0	X'00000000'
MQRO_DEAD_LETTER_Q	0	X'00000000'
MQRO_NONE	0	X'00000000'
MQRO_PAN	1	X'00000001'
MQRO_NAN	2	X'00000002'
MQRO_PASS_CORREL_ID	64	X'00000040'
MQRO_PASS_MSG_ID	128	X'00000080'
MQRO_COA	256	X'00000100'
MQRO_COA_WITH_DATA	768	X'00000300'
MQRO_COA_WITH_FULL_DATA	1792	X'00000700'
MQRO_COD	2048	X'00000800'
MQRO_COD_WITH_DATA	6144	X'00001800'
MQRO_COD_WITH_FULL_DATA	14336	X'00003800'
MQRO_EXPIRATION	2097152	X'00200000'
MQRO_EXPIRATION_WITH_DATA	6291456	X'00600000'
MQRO_EXPIRATION_WITH_FULL_DATA	14680064	X'00E00000'

```
MQRO_EXCEPTION                    16777216        X'01000000'
MQRO_EXCEPTION_WITH_DATA          50331648        X'03000000'
MQRO_EXCEPTION_WITH_FULL_DATA    117440512        X'07000000'
MQRO_DISCARD_MSG                 134217728        X'08000000'
```

MQRO_★ (Report-options masks)

See Appendix C, “Report options and message flags” on page 489.

```
MQRO_REJECT_UNSUP_MASK          270270464        X'101C0000'
MQRO_ACCEPT_UNSUP_MASK         -270532353        X'EFE000FF'
MQRO_ACCEPT_UNSUP_IF_XMIT_MASK  261888        X'0003FF00'
```

MQSCO_★ (Queue scope)

See the *Scope* attribute described in “Attributes for all queues” on page 343.

```
MQSCO_Q_MGR                           1        X'00000001'
MQSCO_CELL                            2        X'00000002'
```

MQSEG_★ (Segmentation)

See the *Segmentation* field described in “MQGMO – Get-message options” on page 56.

```
MQSEG_INHIBITED                     'b'
MQSEG_ALLOWED                       'A'
```

MQSP_★ (Syncpoint)

See the *SyncPoint* attribute described in “Attributes for the queue manager” on page 370.

```
MQSP_NOT_AVAILABLE                    0        X'00000000'
MQSP_AVAILABLE                        1        X'00000001'
```

MQSS_★ (Segment status)

See the *SegmentStatus* field described in “MQGMO – Get-message options” on page 56.

```
MQSS_NOT_A_SEGMENT                  'b'
MQSS_LAST_SEGMENT                   'L'
MQSS_SEGMENT                        'S'
```

MQTC_★ (Trigger control)

See the *TriggerControl* attribute described in “Attributes for local queues and model queues” on page 348.

```
MQTC_OFF                              0        X'00000000'
MQTC_ON                               1        X'00000001'
```

MQTM_★ (Trigger message structure identifier)

See the *StrucId* field described in "MQTM – Trigger message" on page 209.

MQTM_STRUC_ID	'TMbb'

For the C programming language, the following is also defined:

MQTM_STRUC_ID_ARRAY	'T','M','b','b'

MQTM_★ (Trigger message version)

See the *Version* field described in "MQTM – Trigger message" on page 209.

MQTM_VERSION_1	1	X'00000001'
MQTM_CURRENT_VERSION	1	X'00000001'

MQTMC_★ (Trigger message character format structure identifier)

See the *StrucId* field described in "MQTMC2 – Trigger message 2 (character format)" on page 217.

MQTMC_STRUC_ID	'TMCb'

For the C programming language, the following is also defined:

MQTMC_STRUC_ID_ARRAY	'T','M','C','b'

MQTMC_★ (Trigger message character format version)

See the *Version* field described in "MQTMC2 – Trigger message 2 (character format)" on page 217.

MQTMC_VERSION_1	'bbb1'
MQTMC_VERSION_2	'bbb2'
MQTMC_CURRENT_VERSION	'bbb2'

For the C programming language, the following are also defined:

MQTMC_VERSION_1_ARRAY	'b','b','b','1'
MQTMC_VERSION_2_ARRAY	'b','b','b','2'
MQTMC_CURRENT_VERSION_ARRAY	'b','b','b','2'

MQTT_★ (Trigger type)

See the *TriggerType* attribute described in "Attributes for local queues and model queues" on page 348.

MQTT_NONE	0	X'00000000'
MQTT_FIRST	1	X'00000001'
MQTT_EVERY	2	X'00000002'
MQTT_DEPTH	3	X'00000003'

MQUS_★ (Usage)

See the *Usage* attribute described in "Attributes for local queues and model queues" on page 348.

MQUS_NORMAL	0	X'00000000'
MQUS_TRANSMISSION	1	X'00000001'

MQWI_★ (Wait interval)

See the *WaitInterval* field described in "MQGMO – Get-message options" on page 56.

MQWI_UNLIMITED	-1	X'FFFFFFFF'

MQXC_★ (Exit command identifier)

See the *ExitCommand* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222.

MQXC_MQOPEN	1	X'00000001'
MQXC_MQCLOSE	2	X'00000002'
MQXC_MQGET	3	X'00000003'
MQXC_MQPUT	4	X'00000004'
MQXC_MQPUT1	5	X'00000005'
MQXC_MQINQ	6	X'00000006'
MQXC_MQSET	8	X'00000008'
MQXC_MQBACK	9	X'00000009'
MQXC_MQCMIT	10	X'0000000A'

MQXCC_★ (Exit response)

See the *ExitResponse* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222, and the description of the MQCXP structure in the *MQSeries Intercommunication* book.

MQXCC_CLOSE_CHANNEL	-6	X'FFFFFFFA'
MQXCC_SUPPRESS_EXIT	-5	X'FFFFFFFB'
MQXCC_SEND_SEC_MSG	-4	X'FFFFFFFC'
MQXCC_SEND_AND_REQUEST_SEC_MSG	-3	X'FFFFFFFD'
MQXCC_SKIP_FUNCTION	-2	X'FFFFFFFE'
MQXCC_SUPPRESS_FUNCTION	-1	X'FFFFFFFF'
MQXCC_OK	0	X'00000000'

MQXDR_★ (Data-conversion-exit response)

See the *ExitResponse* field described in "MQDXP – Data-conversion exit parameter structure" on page 502.

MQXDR_OK	0	X'00000000'
MQXDR_CONVERSION_FAILED	1	X'00000001'

MQXP_★ (Exit parameter block structure identifier)

See the *StrucId* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222.

MQXP_STRUC_ID	'XPbb'

For the C programming language, the following is also defined:

MQXP_STRUC_ID_ARRAY	'X','P','b','b'

MQXP_★ (Exit parameter block version)

See the *Version* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222.

MQXP_VERSION_1	1	X'00000001'

MQXQH_★ (Transmission queue header structure identifier)

See the *StrucId* field described in "MQXQH – Transmission queue header" on page 227.

MQXQH_STRUC_ID	'XQHb'

For the C programming language, the following is also defined:

MQXQH_STRUC_ID_ARRAY	'X','Q','H','b'

MQXQH_★ (Transmission queue header version)

See the *Version* field described in "MQXQH – Transmission queue header" on page 227.

MQXQH_VERSION_1	1	X'00000001'
MQXQH_CURRENT_VERSION	1	X'00000001'

MQXR_★ (Exit reason)

See the *ExitReason* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222, and the description of the MQCXP structure in the *MQSeries Intercommunication* book.

MQXR_BEFORE	1	X'00000001'
MQXR_AFTER	2	X'00000002'

MQXT_★ (Exit identifier)

See the *ExitId* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222, and the description of the MQCXP structure in the *MQSeries Intercommunication* book.

MQXT_API_CROSSING_EXIT	1	X'00000001'
MQXT_CHANNEL_SEC_EXIT	11	X'0000000B'
MQXT_CHANNEL_MSG_EXIT	12	X'0000000C'
MQXT_CHANNEL_SEND_EXIT	13	X'0000000D'
MQXT_CHANNEL_RCV_EXIT	14	X'0000000E'
MQXT_CHANNEL_MSG_RETRY_EXIT	15	X'0000000F'

MQXT_CHANNEL_AUTO_DEF_EXIT	16	X'00000010'

MQXUA_★ (Exit user area)

See the *ExitUserArea* field described in "MQXP – Exit parameter block (MVS/ESA only)" on page 222, and the description of the MQCXP structure in the *MQSeries Intercommunication* book.

MQXUA_NONE	X'00...00' (16 nulls)

For the C programming language, the following is also defined:

MQXUA_NONE_ARRAY	'\0','\0',...'\0','\0'

Appendix A. Rules for validating MQI options

This appendix explains the situations that produce an MQRC_OPTIONS_ERROR reason code from an MQOPEN, MQPUT, MQPUT1, MQGET, or MQCLOSE call.

MQOPEN

For the options of the MQOPEN call:

- Only valid options are allowed.
- At least *one* of the following must be specified:
  - MQOO_INPUT_EXCLUSIVE
  - MQOO_INPUT_SHARED
  - MQOO_INPUT_AS_Q_DEF
  - MQOO_BROWSE
  - MQOO_OUTPUT
  - MQOO_INQUIRE
  - MQOO_SET
- Only *one* of the following is allowed:
  - MQOO_INPUT_EXCLUSIVE
  - MQOO_INPUT_SHARED
  - MQOO_INPUT_AS_Q_DEF
- If MQOO_SAVE_ALL_CONTEXT is specified, one of the MQOO_INPUT_⋆ options must also be specified.
- If one of the MQOO_SET_⋆_CONTEXT or MQOO_PASS_⋆_CONTEXT options is specified, MQOO_OUTPUT must also be specified.

MQPUT

For the put-message options:

- Only valid options are allowed.
- The combination of MQPMO_SYNCPOINT and MQPMO_NO_SYNCPOINT is not allowed.
- Only *one* of the following is allowed:
  - MQPMO_NO_CONTEXT
  - MQPMO_DEFAULT_CONTEXT
  - MQPMO_PASS_IDENTITY_CONTEXT
  - MQPMO_PASS_ALL_CONTEXT
  - MQPMO_SET_IDENTITY_CONTEXT
  - MQPMO_SET_ALL_CONTEXT
- MQPMO_ALTERNATE_USER_AUTHORITY is not allowed (it is valid only on the MQPUT1 call).

MQPUT1

For the put-message options, the rules are the same as for the MQPUT call, except that MQPMO_ALTERNATE_USER_AUTHORITY is allowed.

MQGET

For the get-message options:

- Only valid options are allowed.
- Only *one* of the following is allowed:
  - MQGMO_SYNCPOINT
  - MQGMO_SYNCPOINT_IF_PERSISTENT
  - MQGMO_NO_SYNCPOINT
- Only *one* of the following is allowed:
  - MQGMO_BROWSE_FIRST
  - MQGMO_BROWSE_NEXT
  - MQGMO_BROWSE_MSG_UNDER_CURSOR
  - MQGMO_MSG_UNDER_CURSOR
- MQGMO_SYNCPOINT is not allowed with any of the following:

  MQGMO_BROWSE_FIRST
  MQGMO_BROWSE_NEXT
  MQGMO_BROWSE_MSG_UNDER_CURSOR
  MQGMO_LOCK
  MQGMO_UNLOCK
- MQGMO_SYNCPOINT_IF_PERSISTENT is not allowed with any of the following:

  MQGMO_BROWSE_FIRST
  MQGMO_BROWSE_NEXT
  MQGMO_BROWSE_MSG_UNDER_CURSOR
  MQGMO_COMPLETE_MSG
  MQGMO_LOCK
  MQGMO_UNLOCK
- MQGMO_MARK_SKIP_BACKOUT requires MQGMO_SYNCPOINT to be specified.
- The combination of MQGMO_WAIT and MQGMO_SET_SIGNAL is not allowed.
- If MQGMO_LOCK is specified, one of the following must also be specified:

  MQGMO_BROWSE_FIRST
  MQGMO_BROWSE_NEXT
  MQGMO_BROWSE_MSG_UNDER_CURSOR
- If MQGMO_UNLOCK is specified, only the following are allowed:

  MQGMO_NO_WAIT
  MQGMO_NO_SYNCPOINT

MQCLOSE

For the options of the MQCLOSE call:

- Only valid options are allowed.
- The combination of MQCO_DELETE and MQCO_DELETE_PURGE is not allowed.

Appendix B. Machine encodings

This appendix describes the structure of the *Encoding* field in the message descriptor MQMD (see page 118).

The *Encoding* field is a 32-bit integer that is divided into four separate subfields; these subfields identify:

- The encoding used for binary integers
- The encoding used for packed-decimal integers
- The encoding used for floating-point numbers
- Reserved bits

Each subfield is identified by a bit mask which has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined:

MQENC_INTEGER_MASK
: Mask for binary-integer encoding.

  This subfield occupies bit positions 28 through 31 within the *Encoding* field.

MQENC_DECIMAL_MASK
: Mask for packed-decimal-integer encoding.

  This subfield occupies bit positions 24 through 27 within the *Encoding* field.

MQENC_FLOAT_MASK
: Mask for floating-point encoding.

  This subfield occupies bit positions 20 through 23 within the *Encoding* field.

MQENC_RESERVED_MASK
: Mask for reserved bits.

  This subfield occupies bit positions 0 through 19 within the *Encoding* field.

Binary-integer encoding

The following values are valid for the binary-integer encoding:

MQENC_INTEGER_UNDEFINED
: Undefined integer encoding.

  Binary integers are represented using an encoding that is undefined.

MQENC_INTEGER_NORMAL
: Normal integer encoding.

  Binary integers are represented in the conventional way:

  - The least significant byte in the number has the highest address of any of the bytes in the number; the most significant byte has the lowest address
  - The least significant bit in each byte is adjacent to the byte with the next higher address; the most significant bit in each byte is adjacent to the byte with the next lower address

MQENC_INTEGER_REVERSED
Reversed integer encoding.

Binary integers are represented in the same way as MQENC_INTEGER_NORMAL, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as MQENC_INTEGER_NORMAL.

Packed-decimal-integer encoding

The following values are valid for the packed-decimal-integer encoding:

MQENC_DECIMAL_UNDEFINED
Undefined packed-decimal encoding.

Packed-decimal integers are represented using an encoding that is undefined.

MQENC_DECIMAL_NORMAL
Normal packed-decimal encoding.

Packed-decimal integers are represented in the conventional way:

- Each decimal digit in the printable form of the number is represented in packed decimal by a single hexadecimal digit in the range `X'0'` through `X'9'`. Each hexadecimal digit occupies four bits, and so each byte in the packed decimal number represents two decimal digits in the printable form of the number.
- The least significant byte in the packed-decimal number is the byte which contains the least significant decimal digit. Within that byte, the most significant four bits contain the least significant decimal digit, and the least significant four bits contain the sign. The sign is either `X'C'` (positive), `X'D'` (negative), or `X'F'` (unsigned).
- The least significant byte in the number has the highest address of any of the bytes in the number; the most significant byte has the lowest address.
- The least significant bit in each byte is adjacent to the byte with the next higher address; the most significant bit in each byte is adjacent to the byte with the next lower address.

MQENC_DECIMAL_REVERSED
Reversed packed-decimal encoding.

Packed-decimal integers are represented in the same way as MQENC_DECIMAL_NORMAL, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as MQENC_DECIMAL_NORMAL.

Floating-point encoding

The following values are valid for the floating-point encoding:

MQENC_FLOAT_UNDEFINED
Undefined floating-point encoding.

Floating-point numbers are represented using an encoding that is undefined.

MQENC_FLOAT_IEEE_NORMAL
Normal IEEE float encoding.

Floating-point numbers are represented using the standard IEEE[5] floating-point format, with the bytes arranged as follows:

- The least significant byte in the mantissa has the highest address of any of the bytes in the number; the byte containing the exponent has the lowest address
- The least significant bit in each byte is adjacent to the byte with the next higher address; the most significant bit in each byte is adjacent to the byte with the next lower address

Details of the IEEE float encoding may be found in IEEE Standard 754.

MQENC_FLOAT_IEEE_REVERSED
Reversed IEEE float encoding.

Floating-point numbers are represented in the same way as MQENC_FLOAT_IEEE_NORMAL, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as MQENC_FLOAT_IEEE_NORMAL.

MQENC_FLOAT_S390
System/390 architecture float encoding.

Floating-point numbers are represented using the standard System/390 floating-point format; this is also used by System/370.

Constructing encodings

To construct a value for the *Encoding* field in MQMD, the relevant constants that describe the required encodings can be:

- Added together, or
- Combined using the bitwise OR operation (if the programming language supports bit operations)

Whichever method is used, combine only one of the MQENC_INTEGER_★ encodings with one of the MQENC_DECIMAL_★ encodings and one of the MQENC_FLOAT_★ encodings.

Analyzing encodings

The *Encoding* field contains subfields; because of this, applications that need to examine the integer, packed decimal, or float encoding should use one of the techniques described below.

Using bit operations

If the programming language supports bit operations, the following steps should be performed:

1. Select one of the following values, according to the type of encoding required:

Encoding	Value to use
Binary integer	MQENC_INTEGER_MASK
Packed-decimal integer	MQENC_DECIMAL_MASK

[5] The Institute of Electrical and Electronics Engineers

Floating point	MQENC_FLOAT_MASK

Call the value A.

2. Combine the *Encoding* field with A using the bitwise AND operation; call the result B.
3. B is the encoding required, and can be tested for equality with each of the values that is valid for that type of encoding.

Using arithmetic

If the programming language *does not* support bit operations, the following steps should be performed using integer arithmetic:

1. Select a value from the following table, according to the encoding required:

Encoding required	Value to use
Binary integer	1
Packed-decimal integer	16
Floating point	256

Call the value A.

2. Divide the value of the *Encoding* field by A; call the result B.
3. Divide B by 16; call the result C.
4. Multiply C by 16 and subtract from B; call the result D.
5. Multiply D by A; call the result E.
6. E is the encoding required, and can be tested for equality with each of the values that is valid for that type of encoding.

Summary of machine architecture encodings

Encodings for machine architectures are shown in Table 67.

Table 67. Summary of encodings for machine architectures

Machine architecture	Binary integer encoding	Packed-decimal integer encoding	Floating-point encoding
AS/400	normal	normal	IEEE normal
Intel x86	reversed	reversed	IEEE reversed
PowerPC	normal	normal	IEEE normal
System/390	normal	normal	System/390

Appendix C. Report options and message flags

This appendix concerns the *Report* and *MsgFlags* fields that are part of the message descriptor MQMD specified on the MQGET, MQPUT, and MQPUT1 calls (see page 101). The appendix describes:

- The structure of the report field and how the queue manager processes it
- How an application should analyze the report field
- The structure of the message-flags field

Structure of the report field

The *Report* field is a 32-bit integer that is divided into three separate subfields. These subfields identify:

- Report options that are rejected if the local queue manager does not recognize them
- Report options that are always accepted, even if the local queue manager does not recognize them
- Report options that are accepted only if certain other conditions are satisfied

Each subfield is identified by a bit mask which has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. Note that the bits in a subfield are not necessarily adjacent. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

MQRO_REJECT_UNSUP_MASK
: Mask for unsupported report options that are rejected.

  This mask identifies the bit positions within the *Report* field where report options which are not supported by the local queue manager will cause the MQPUT or MQPUT1 call to fail with completion code MQCC_FAILED and reason code MQRC_REPORT_OPTIONS_ERROR.

  This subfield occupies bit positions 3, and 11 through 13.

MQRO_ACCEPT_UNSUP_MASK
: Mask for unsupported report options that are accepted.

  This mask identifies the bit positions within the *Report* field where report options which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls. Completion code MQCC_WARNING with reason code MQRC_UNKNOWN_REPORT_OPTION are returned in this case.

  This subfield occupies bit positions 0 through 2, 4 through 10, and 24 through 31.

  The following report options are included in this subfield:

  MQRO_COPY_MSG_ID_TO_CORREL_ID
  MQRO_DEAD_LETTER_Q
  MQRO_DISCARD_MSG
  MQRO_EXCEPTION
  MQRO_EXCEPTION_WITH_DATA

MQRO_EXCEPTION_WITH_FULL_DATA
MQRO_EXPIRATION
MQRO_EXPIRATION_WITH_DATA
MQRO_EXPIRATION_WITH_FULL_DATA
MQRO_NAN
MQRO_NEW_MSG_ID
MQRO_NONE
MQRO_PAN
MQRO_PASS_CORREL_ID
MQRO_PASS_MSG_ID

MQRO_ACCEPT_UNSUP_IF_XMIT_MASK
Mask for unsupported report options that are accepted only in certain circumstances.

This mask identifies the bit positions within the *Report* field where report options which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls *provided* that both of the following conditions are satisfied:

- The message is destined for a remote queue manager.
- The application is not putting the message directly on a local transmission queue (that is, the queue identified by the *ObjectQMgrName* and *ObjectName* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code MQCC_WARNING with reason code MQRC_UNKNOWN_REPORT_OPTION are returned if these conditions are satisfied, and MQCC_FAILED with reason code MQRC_REPORT_OPTIONS_ERROR if not.

This subfield occupies bit positions 14 through 23.

The following report options are included in this subfield:

MQRO_COA
MQRO_COA_WITH_DATA
MQRO_COA_WITH_FULL_DATA
MQRO_COD
MQRO_COD_WITH_DATA
MQRO_COD_WITH_FULL_DATA

If there are any options specified in the *Report* field which the queue manager does not recognize, the queue manager checks each subfield in turn by using the bitwise AND operation to combine the *Report* field with the mask for that subfield. If the result of that operation is not zero, the completion code and reason codes described above are returned.

If MQCC_WARNING is returned, it is not defined which reason code is returned if other warning conditions exist.

The ability to specify and have accepted report options which are not recognized by the local queue manager is useful when it is desired to send a message with a report option which will be recognized and processed by a *remote* queue manager.

Analyzing the report field

The *Report* field contains subfields; because of this, applications that need to check whether the sender of the message requested a particular report should use one of the techniques described below.

Using bit operations

If the programming language supports bit operations, the following steps should be performed:

1. Select one of the following values, according to the type of report to be checked:

Report type	Value to use
COA	MQRO_COA_WITH_FULL_DATA
COD	MQRO_COD_WITH_FULL_DATA
Exception	MQRO_EXCEPTION_WITH_FULL_DATA
Expiration	MQRO_EXPIRATION_WITH_FULL_DATA

   Call the value A.

   On MVS/ESA, the MQRO_*_WITH_DATA values should be used instead of the MQRO_*_WITH_FULL_DATA values.

2. Combine the *Report* field with A using the bitwise AND operation; call the result B.

3. Test B for equality with each of the values that is possible for that type of report.

   For example, if A is MQRO_EXCEPTION_WITH_FULL_DATA, test B for equality with each of the following to determine what was specified by the sender of the message:

   MQRO_NONE
   MQRO_EXCEPTION
   MQRO_EXCEPTION_WITH_DATA
   MQRO_EXCEPTION_WITH_FULL_DATA

   The tests can be performed in whatever order is most convenient for the application logic.

A similar method can be used to test for the MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID options; select as the value A whichever of these two constants is appropriate, and then proceed as described above.

Using arithmetic

If the programming language *does not* support bit operations, the following steps should be performed using integer arithmetic:

1. Select one of the following values, according to the type of report to be checked:

Report type	Value to use
COA	MQRO_COA
COD	MQRO_COD
Exception	MQRO_EXCEPTION
Expiration	MQRO_EXPIRATION

Call the value A.

2. Divide the *Report* field by A; call the result B.
3. Divide B by 8; call the result C.
4. Multiply C by 8 and subtract from B; call the result D.
5. Multiply D by A; call the result E.
6. Test E for equality with each of the values that is possible for that type of report.

   For example, if A is MQRO_EXCEPTION, test E for equality with each of the following to determine what was specified by the sender of the message:

   MQRO_NONE
   MQRO_EXCEPTION
   MQRO_EXCEPTION_WITH_DATA
   MQRO_EXCEPTION_WITH_FULL_DATA

   The tests can be performed in whatever order is most convenient for the application logic.

The following pseudocode illustrates this technique for exception report messages:

```
A = MQRO_EXCEPTION
B = Report/A
C = B/8
D = B - C*8
E = D*A
```

A similar method can be used to test for the MQRO_PASS_MSG_ID or MQRO_PASS_CORREL_ID options; select as the value A whichever of these two constants is appropriate, and then proceed as described above, but replacing the value 8 in the steps above by the value 2.

Structure of the message-flags field

The *MsgFlags* field is a 32-bit integer that is divided into three separate subfields. These subfields identify:

- Message flags that are rejected if the local queue manager does not recognize them
- Message flags that are always accepted, even if the local queue manager does not recognize them
- Message flags that are accepted only if certain other conditions are satisfied

Note: All subfields in *MsgFlags* are reserved for use by the queue manager.

Each subfield is identified by a bit mask which has 1-bits in the positions corresponding to the subfield, and 0-bits elsewhere. The bits are numbered such that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

MQMF_REJECT_UNSUP_MASK
: Mask for unsupported message flags that are rejected.

  This mask identifies the bit positions within the *MsgFlags* field where message flags which are not supported by the local queue manager will cause the

MQPUT or MQPUT1 call to fail with completion code MQCC_FAILED and reason code MQRC_MSG_FLAGS_ERROR.

This subfield occupies bit positions 20 through 31.

The following message flags are included in this subfield:

MQMF_LAST_MSG_IN_GROUP
MQMF_LAST_SEGMENT
MQMF_MSG_IN_GROUP
MQMF_SEGMENT
MQMF_SEGMENTATION_ALLOWED

MQMF_ACCEPT_UNSUP_MASK
Mask for unsupported message flags that are accepted.

This mask identifies the bit positions within the *MsgFlags* field where message flags which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls. The completion code is MQCC_OK.

This subfield occupies bit positions 0 through 11.

MQMF_ACCEPT_UNSUP_IF_XMIT_MASK
Mask for unsupported message flags that are accepted only in certain circumstances.

This mask identifies the bit positions within the *MsgFlags* field where message flags which are not supported by the local queue manager will nevertheless be accepted on the MQPUT or MQPUT1 calls *provided* that both of the following conditions are satisfied:

- The message is destined for a remote queue manager.
- The application is not putting the message directly on a local transmission queue (that is, the queue identified by the *ObjectQMgrName* and *ObjectName* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code MQCC_OK is returned if these conditions are satisfied, and MQCC_FAILED with reason code MQRC_MSG_FLAGS_ERROR if not.

This subfield occupies bit positions 12 through 19.

If there are flags specified in the *MsgFlags* field that the queue manager does not recognize, the queue manager checks each subfield in turn by using the bitwise AND operation to combine the *MsgFlags* field with the mask for that subfield. If the result of that operation is not zero, the completion code and reason codes described above are returned.

Appendix D. Data-conversion

This appendix describes the interface to the data-conversion exit, and the processing performed by the queue manager when data conversion is required.

The data-conversion exit is invoked as part of the processing of the MQGET call, in order to convert the application message data to the representation required by the receiving application. Conversion of the application message data is optional — it requires the MQGMO_CONVERT option to be specified on the MQGET call.

The following are described:

- The processing performed by the queue manager in response to the MQGMO_CONVERT option; see "Conversion processing."
- Processing conventions used by the queue manager when processing a built-in format; these conventions are recommended for user-written exits too. See "Processing conventions" on page 497.
- Special considerations for the conversion of report messages; see "Conversion of report messages" on page 501.
- The parameters passed to the data-conversion exit; see "MQDATACONVEXIT – Data conversion exit" on page 515.
- A call that can be used from the exit in order to convert character data between different representations; see "MQXCNVC – Convert characters" on page 509.
- The data-structure parameter which is specific to the exit; see "MQDXP – Data-conversion exit parameter structure" on page 502.

Conversion processing

The queue manager performs the following actions if the MQGMO_CONVERT option is specified on the MQGET call, and there is a message to be returned to the application:

1. If one or more of the following is true, no conversion is necessary:
   - The *CodedCharSetId* and *Encoding* values in the control information in the message are identical to those in the *MsgDesc* parameter.
   - The length of the application message data is zero.
   - The length of the *Buffer* parameter is zero.

   In these cases the message is returned without conversion to the application issuing the MQGET call; the *CodedCharSetId* and *Encoding* values in the *MsgDesc* parameter are set to the values in the control information in the message, and the call completes with one of the following combinations of completion code and reason code:

Completion code	Reason code
MQCC_OK	MQRC_NONE
MQCC_WARNING	MQRC_TRUNCATED_MSG_ACCEPTED
MQCC_WARNING	MQRC_TRUNCATED_MSG_FAILED

The following steps are performed only if the *CodedCharSetId* or *Encoding* value in the control information in the message differs from that in the *MsgDesc* parameter, and there is data to be converted:

2. If the *Format* field in the control information in the message has the value MQFMT_NONE, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

   In all other cases conversion processing continues.

3. The message is removed from the queue and placed in a temporary buffer which is the same size as the *Buffer* parameter. For browse operations, the message is copied into the temporary buffer, instead of being removed from the queue.

4. If the message has to be truncated to fit in the buffer, the following is done:

   - If the MQGMO_ACCEPT_TRUNCATED_MSG option was *not* specified, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_TRUNCATED_MSG_FAILED.
   - If the MQGMO_ACCEPT_TRUNCATED_MSG option *was* specified, the completion code is set to MQCC_WARNING, the reason code is set to MQRC_TRUNCATED_MSG_ACCEPTED, and conversion processing continues.

5. If the message can be accommodated in the buffer without truncation, or the MQGMO_ACCEPT_TRUNCATED_MSG option was specified, the following is done:

   - If the format is a built-in format, the buffer is passed to the queue-manager's data-conversion service.
   - If the format is not a built-in format, the buffer is passed to a user-written exit which has the same name as the format. If the exit cannot be found, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_FORMAT_ERROR.

   If no error occurs, the output from the data-conversion service or from the user-written exit is the converted message, plus the completion code and reason code to be returned to the application issuing the MQGET call.

6. If the conversion is successful, the queue manager returns the converted message to the application. In this case, the completion code and reason code returned will usually be one of the following combinations:

Completion code	Reason code
MQCC_OK	MQRC_NONE
MQCC_WARNING	MQRC_TRUNCATED_MSG_ACCEPTED

   If the conversion fails (for whatever reason), the queue manager returns the unconverted message to the application, with the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter set to the values in the control information in the message, and with completion code MQCC_WARNING. See below for possible reason codes.

Processing conventions

When converting a built-in format, the queue manager follows the processing conventions described below. It is recommended that user-written exits should also follow these conventions, although this is not enforced by the queue manager. The built-in formats converted by the queue manager are:

MQFMT_ADMIN
MQFMT_COMMAND_1
MQFMT_COMMAND_2
MQFMT_DEAD_LETTER_HEADER
MQFMT_DIST_HEADER
MQFMT_EVENT
MQFMT_IMS
MQFMT_IMS_VAR_STRING
MQFMT_MD_EXTENSION
MQFMT_PCF
MQFMT_REF_MSG_HEADER
MQFMT_STRING
MQFMT_TRIGGER
MQFMT_XMIT_Q_HEADER

1. If the message expands during conversion, and exceeds the size of the *Buffer* parameter, the following is done:
   - If the MQGMO_ACCEPT_TRUNCATED_MSG option was *not* specified, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_CONVERTED_MSG_TOO_BIG.
   - If the MQGMO_ACCEPT_TRUNCATED_MSG option *was* specified, the message is truncated, the completion code is set to MQCC_WARNING, the reason code is set to MQRC_TRUNCATED_MSG_ACCEPTED, and conversion processing continues.
2. If truncation occurs (either before or during conversion), it is possible for the number of valid bytes returned in the *Buffer* parameter to be *less than* the length of the buffer.

   This can occur, for example, if a 4-byte integer or a DBCS character straddles the end of the buffer. The incomplete element of information is not converted, and so those bytes in the returned message do not contain valid information. This can also occur if a message that was truncated before conversion shrinks during conversion.

   If the number of valid bytes returned is less than the length of the buffer, the unused bytes at the end of the buffer are set to nulls.
3. If an array or string straddles the end of the buffer, as much of the data as possible is converted; only the particular array element or DBCS character which is incomplete is not converted – preceding array elements or characters are converted.
4. If truncation occurs (either before or during conversion), the length returned for the *DataLength* parameter is the length of the *unconverted* message before truncation.
5. The data returned to the application is never partially converted; either all of the data returned is converted, or none of it is. For example, if the integers in the

data can be converted, but the character strings cannot (because the character-set identifier is not recognized), none of the data is converted.

6. If the *CodedCharSetId* or *Encoding* fields in the control information of the message being retrieved, or in the *MsgDesc* parameter, specify values which are undefined or not supported, the queue manager may ignore the error if the undefined or unsupported value does not need to be used in converting the message.

   For example, if the *Encoding* field in the message specifies an unsupported float encoding, but the message contains only integer data, or contains floating-point data which does not require conversion (because the source and target float encodings are identical), the error may or may not be diagnosed.

   If the error is diagnosed, the message is returned unconverted, with completion code MQCC_WARNING and one of the MQRC_SOURCE_*_ERROR or MQRC_TARGET_*_ERROR reason codes (as appropriate); the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to the values in the control information in the message.

   If the error is not diagnosed and the conversion completes successfully, the values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are those specified by the application issuing the MQGET call.

7. In all cases, if the message is returned to the application unconverted the completion code is set to MQCC_WARNING, and the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter are set to the values appropriate to the unconverted data. This is done for MQFMT_NONE also.

   The *Reason* parameter is set to a code that indicates why the conversion could not be carried out, unless the message also had to be truncated; reason codes related to truncation take precedence over reason codes related to conversion. (To determine if a truncated message was converted, check the values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter.)

   When an error is diagnosed, either a specific reason code is returned, or the general reason code MQRC_NOT_CONVERTED. The reason code returned depends on the diagnostic capabilities of the underlying data-conversion service.

8. If completion code MQCC_WARNING is returned, and more than one reason code is relevant, the order of precedence is as follows:

   a. The following reasons take precedence over all others; only one of the reasons in this group can arise:

      MQRC_SIGNAL_REQUEST_ACCEPTED
      MQRC_TRUNCATED_MSG_ACCEPTED

   b. Next in precedence is the following reason:

      MQRC_FORMAT_ERROR

   c. The order of precedence within this final group is not defined:

      MQRC_CONVERTED_MSG_TOO_BIG
      MQRC_NOT_CONVERTED
      MQRC_SOURCE_CCSID_ERROR
      MQRC_SOURCE_DECIMAL_ENC_ERROR
      MQRC_SOURCE_FLOAT_ENC_ERROR
      MQRC_SOURCE_INTEGER_ENC_ERROR

MQRC_TARGET_CCSID_ERROR
MQRC_TARGET_DECIMAL_ENC_ERROR
MQRC_TARGET_FLOAT_ENC_ERROR
MQRC_TARGET_INTEGER_ENC_ERROR

9. On completion of the MQGET call:
   - The following reason code indicates that the message was converted successfully:

     MQRC_NONE

   - The following reason code indicates that the message *may* have been converted successfully (check the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to find out):

     MQRC_TRUNCATED_MSG_ACCEPTED

   - All other reason codes indicate that the message was not converted.

The following processing is specific to the built-in formats; it is not applicable to user-defined formats:

10. With the exception of the format MQFMT_STRING, none of the built-in formats can be converted from or to double-byte character sets (DBCS); only single-byte character sets (SBCS) can be used with these formats.

    If DBCS character sets are specified with these formats, the message is returned unconverted, with completion code MQCC_WARNING and reason code MQRC_SOURCE_CCSID_ERROR or MQRC_TARGET_CCSID_ERROR, as appropriate.

11. If the message data for a built-in format is truncated, fields within the message which contain lengths of strings, or counts of elements or structures, are *not* adjusted to reflect the length of the data actually returned to the application; the values returned for such fields within the message data are the values applicable to the message *prior to truncation*.

    When processing messages such as a truncated MQFMT_ADMIN message, care must be taken to ensure that the application does not attempt to access data beyond the end of the data returned.

12. If the format name is MQFMT_DEAD_LETTER_HEADER, the message data begins with an MQDLH structure, and this may be followed by zero or more bytes of application message data. The format, character set, and encoding of the application message data are defined by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQDLH structure at the start of the message. Since the MQDLH structure and application message data can have different character sets and encodings, it is possible for one, other, or both of the MQDLH structure and application message data to require conversion.

    The queue manager converts the MQDLH structure first, as necessary. If conversion is successful, or the MQDLH structure does not require conversion, the queue manager checks the *CodedCharSetId* and *Encoding* fields in the MQDLH structure to see if conversion of the application message data is required. If conversion *is* required, the queue manager invokes the user-written exit with the name given by the *Format* field in the MQDLH structure, or performs the conversion itself (if *Format* is the name of a built-in format).

If the MQGET call returns a completion code of MQCC_WARNING, and the reason code is one of those indicating that conversion was not successful, one of the following applies:

- The MQDLH structure could not be converted. In this case the application message data will not have been converted either.
- The MQDLH structure was converted, but the application message data was not.

The application can examine the values returned in the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter, and those in the MQDLH structure, in order to determine which of the above applies.

13. If the format name is MQFMT_XMIT_Q_HEADER, the message data begins with an MQXQH structure, and this may be followed by zero or more bytes of additional data. This additional data is usually the application message data (which may be of zero length), but there can also be one or more further MQ header structures present, at the start of the additional data.

    The MQXQH structure must be in the character set and encoding of the queue manager. The format, character set, and encoding of the data following the MQXQH structure are given by the *Format*, *CodedCharSetId*, and *Encoding* fields in the MQMD structure contained *within* the MQXQH. For each subsequent MQ header structure present, the *Format*, *CodedCharSetId*, and *Encoding* fields in the structure describe the data that follows that structure; that data is either another MQ header structure, or the application message data.

    If the MQGMO_CONVERT option is specified for an MQFMT_XMIT_Q_HEADER message, the application message data and certain of the MQ header structures are converted, *but the data in the MQXQH structure is not*. On return from the MQGET call, therefore:

    - The values of the *Format*, *CodedCharSetId*, and *Encoding* fields in the *MsgDesc* parameter describe the data in the MQXQH structure, and *not* the application message data; the values will therefore *not* be the same as those specified by the application that issued the MQGET call.

      The effect of this is that an application which repeatedly gets messages from a transmission queue with the MQGMO_CONVERT option specified must reset the *CodedCharSetId* and *Encoding* fields in the *MsgDesc* parameter to the values desired for the application message data, prior to each MQGET call.

    - The values of the *Format*, *CodedCharSetId*, and *Encoding* fields in the last MQ header structure present describe the application message data. If there are no other MQ header structures present, the application message data is described by these fields in the MQMD structure within the MQXQH structure. If conversion is successful, the values will be the same as those specified in the *MsgDesc* parameter by the application that issued the MQGET call.

    If the message is a distribution-list message, the MQXQH structure is followed by an MQDH structure (plus its arrays of MQOR and MQPMR records), which in turn may be followed by zero or more further MQ header structures and zero or more bytes of application message data. Like the MQXQH structure, the MQDH structure must be in the character set and encoding of the queue manager, and it is not converted on the MQGET call, even if the MQGMO_CONVERT option is specified.

The processing of the MQXQH and MQDH structures described above is primarily intended for use by message channel agents when they get messages from transmission queues.

Conversion of report messages

A report message can contain varying amounts of application message data, according to the report options specified by the sender of the original message. In particular, a report message can contain either:

1. No application message data
2. Some of the application message data from the original message

   This occurs when the sender of the original message specifies MQRO_*_WITH_DATA and the message is longer than 100 bytes.
3. All of the application message data from the original message

   This occurs when the sender of the original message specifies MQRO_*_WITH_FULL_DATA, or specifies MQRO_*_WITH_DATA and the message is 100 bytes or shorter.

When the queue manager or message channel agent generates a report message, it copies the format name from the original message into the *Format* field in the control information in the report message. The format name in the report message may therefore imply a length of data which is different from the length actually present in the report message (cases 1 and 2 above).

If the MQGMO_CONVERT option is specified when the report message is retrieved:

- For case 1 above, the data-conversion exit will not be invoked (because the report message will have no data).
- For case 3 above, the format name correctly implies the length of the message data.
- But for case 2 above, the data-conversion exit will be invoked to convert a message which is *shorter* than the length implied by the format name.

  In addition, the reason code passed to the exit will usually be MQRC_NONE (that is, the reason code will not indicate that the message has been truncated). This happens because the message data was truncated by the *sender* of the report message, and not by the receiver's queue manager in response to the MQGET call.

Because of these possibilities, the data-conversion exit should *not* use the format name to deduce the length of data passed to it; instead the exit should check the length of data provided, and be prepared to convert *less* data than the length implied by the format name. If the data can be converted successfully, completion code MQCC_OK and reason code MQRC_NONE should be returned by the exit. The length of the message data to be converted is passed to the exit as the *InBufferLength* parameter.

Product-sensitive programming interface

MQDXP – Data-conversion exit parameter structure

The following table summarizes the fields in the structure.

Table 68. Fields in MQDXP

Field	Description	Page
StrucId	Structure identifier	502
Version	Structure version number	503
AppOptions	Application options	503
Encoding	Encoding required by application	503
CodedCharSetId	Character set required by application	503
DataLength	Length in bytes of message data	504
CompCode	Completion code	504
Reason	Reason code qualifying *CompCode*	505
ExitResponse	Response from exit	506
Hconn	Connection handle	507

The MQDXP structure is a parameter that is passed to the data-conversion exit. See the description of the MQDATACONVEXIT call for details of the data conversion exit.

Only the *DataLength*, *CompCode*, *Reason* and *ExitResponse* fields in MQDXP may be changed by the exit; changes to other fields are ignored. However, the *DataLength* field *cannot* be changed if the message being converted is a segment that contains only part of a logical message.

When control returns to the queue manager from the exit, the queue manager checks the values returned in MQDXP. If the values returned are not valid, the queue manager continues processing as though the exit had returned MQXDR_CONVERSION_FAILED in *ExitResponse*; however, the queue manager ignores the values of the *CompCode* and *Reason* fields returned by the exit in this case, and uses instead the values those fields had on *input* to the exit. The following values in MQDXP cause this processing to occur:

- *ExitResponse* field not MQXDR_OK and not MQXDR_CONVERSION_FAILED
- *CompCode* field not MQCC_OK and not MQCC_WARNING
- *DataLength* field less than zero, or *DataLength* field changed when the message being converted is a segment that contains only part of a logical message.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value must be:

MQDXP_STRUC_ID
Identifier for data conversion exit parameter structure.

For the C programming language, the constant MQDXP_STRUC_ID_ARRAY is also defined; this has the same value as MQDXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)
Structure version number.

The value must be:

MQDXP_VERSION_1
Version number for data-conversion exit parameter structure.

The following constant specifies the version number of the current version:

MQDXP_CURRENT_VERSION
Current version of data-conversion exit parameter structure.

Note: When a new version of this structure is introduced, the layout of the existing part is not changed. The exit should therefore check that the *Version* field is equal to or greater than the lowest version which contains the fields that the exit needs to use.

This is an input field to the exit.

ExitOptions (MQLONG)
Reserved.

This is a reserved field; its value is `0`.

AppOptions (MQLONG)
Application options.

This is a copy of the *Options* field of the MQGMO structure specified by the application issuing the MQGET call. The exit may need to examine these to ascertain whether the MQGMO_ACCEPT_TRUNCATED_MSG option was specified.

This is an input field to the exit.

Encoding (MQLONG)
Encoding required by application.

This is the encoding required by the application issuing the MQGET call; see the *Encoding* field in the MQMD structure for more details.

If the conversion is successful, the exit should copy this to the *Encoding* field in the message descriptor.

This is an input field to the exit.

CodedCharSetId (MQLONG)
Character set required by application.

This is the coded character-set identifier of the character set required by the application issuing the MQGET call; see the *CodedCharSetId* field in the MQMD structure for more details. If the application specifies the special value MQCCSI_Q_MGR on the MQGET call, the queue manager

changes this to the actual character-set identifier of the character set used by the queue manager, before invoking the exit.

If the conversion is successful, the exit should copy this to the *CodedCharSetId* field in the message descriptor.

This is an input field to the exit.

DataLength (MQLONG)
Length in bytes of message data.

When the exit is invoked, this field contains the original length of the application message data. If the message was truncated in order to fit into the buffer provided by the application, the size of the message provided to the exit will be *smaller* than the value of *DataLength*. The size of the message actually provided to the exit is always given by the *InBufferLength* parameter of the exit, irrespective of any truncation that may have occurred.

Truncation is indicated by the *Reason* field having the value MQRC_TRUNCATED_MSG_ACCEPTED on input to the exit.

Most conversions will not need to change this length, but an exit can do so if necessary; the value set by the exit is returned to the application in the *DataLength* parameter of the MQGET call. However, this length *cannot* be changed if the message being converted is a segment that contains only part of a logical message. This is because changing the length would cause the offsets of later segments in the logical message to be incorrect.

Note that, if the exit wants to change the length of the data, be aware that the queue manager has already decided whether the message data will fit into the application's buffer, based on the length of the *unconverted* data. This decision determines whether the message is removed from the queue (or the browse cursor moved, for a browse request), and is not affected by any change to the data length caused by the conversion. For this reason it is recommended that conversion exits do not cause a change in the length of the application message data.

If character conversion does imply a change of length, a string can be converted into another string with the same length in bytes, truncating trailing blanks or padding with blanks as necessary.

The exit is not invoked if the message contains no application message data; hence *DataLength* is always greater then zero.

This is an input/output field to the exit.

CompCode (MQLONG)
Completion code.

When the exit is invoked, this contains the completion code that will be returned to the application that issued the MQGET call, if the exit chooses to do nothing. It is always MQCC_WARNING, because either the message was truncated, or the message requires conversion and this has not yet been done.

On output from the exit, this field contains the completion code to be returned to the application in the *CompCode* parameter of the MQGET call; only MQCC_OK and MQCC_WARNING are valid. See the description of

the *Reason* field for recommendations on how the exit should set this field on output.

This is an input/output field to the exit.

Reason (MQLONG)
Reason code qualifying *CompCode*.

When the exit is invoked, this contains the reason code that will be returned to the application that issued the MQGET call, if the exit chooses to do nothing. Among possible values are MQRC_TRUNCATED_MSG_ACCEPTED, indicating that the message was truncated in order fit into the buffer provided by the application, and MQRC_NOT_CONVERTED, indicating that the message requires conversion but that this has not yet been done.

On output from the exit, this field contains the reason to be returned to the application in the *Reason* parameter of the MQGET call; the following is recommended:

- If *Reason* had the value MQRC_TRUNCATED_MSG_ACCEPTED on input to the exit, the *Reason* and *CompCode* fields should not be altered, irrespective of whether the conversion succeeds or fails.

  (If the *CompCode* field is not MQCC_OK, the application which retrieves the message can identify a conversion failure by comparing the returned *Encoding* and *CodedCharSetId* values in the message descriptor with the values requested; in contrast, the application cannot distinguish a truncated message from a message that just fitted the buffer. For this reason, MQRC_TRUNCATED_MSG_ACCEPTED should be returned in preference to any of the reasons that indicate conversion failure.)

- If *Reason* had any other value on input to the exit:
  - If the conversion succeeds, *CompCode* should be set to MQCC_OK and *Reason* set to MQRC_NONE.
  - If the conversion fails, or the message expands and has to be truncated to fit in the buffer, *CompCode* should be set to MQCC_WARNING (or left unchanged), and *Reason* set to one of the values listed below, to indicate the nature of the failure.

    Note that, if the message after conversion is too big for the buffer, it should be truncated only if the application that issued the MQGET call specified the MQGMO_ACCEPT_TRUNCATED_MSG option:

    - If it did specify that option, reason MQRC_TRUNCATED_MSG_ACCEPTED should be returned.
    - If it did not specify that option, the message should be returned unconverted, with reason code MQRC_CONVERTED_MSG_TOO_BIG.

The reason codes listed below are recommended for use by the exit to indicate the reason that conversion failed, but the exit can set other values from the set of MQRC_★ codes if deemed appropriate.

Note: If the message cannot be converted successfully, the exit *must* return MQXDR_CONVERSION_FAILED in the *ExitResponse* field,

in order to cause the queue manager to return the unconverted message. This is true regardless of the reason code returned in the *Reason* field.

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

This is an input/output field to the exit.

ExitResponse (MQLONG)
Response from exit.

This is set by the exit to indicate the success or otherwise of the conversion. It must be one of the following:

MQXDR_OK
Conversion was successful.

If the exit specifies this value, the queue manager returns the following to the application which issued the MQGET call:

- The value of the *CompCode* field on output from the exit
- The value of the *Reason* field on output from the exit
- The value of the *DataLength* field on output from the exit
- The contents of the exit's output buffer *OutBuffer*; the number of bytes returned is the lesser of the exit's *OutBufferLength* parameter, and the value of the *DataLength* field on output from the exit
- The value of the *Encoding* field in the exit's message descriptor parameter on output from the exit

- The value of the *CodedCharSetId* field in the exit's message descriptor parameter on output from the exit

MQXDR_CONVERSION_FAILED
Conversion was unsuccessful.

If the exit specifies this value, the queue manager returns the following to the application which issued the MQGET call:

- The value of the *CompCode* field on output from the exit
- The value of the *Reason* field on output from the exit
- The value of the *DataLength* field on *input* to the exit
- The contents of the exit's input buffer *InBuffer*; the number of bytes returned is given by the *InBufferLength* parameter

If the exit has altered *InBuffer*, the results are undefined.

ExitResponse is an output field from the exit.

Hconn (MQHCONN)
Connection handle.

This is a connection handle which can be used on the MQXCNVC call. This handle is not necessarily the same as the handle specified by the application which issued the MQGET call.

C language declaration

```
typedef struct tagMQDXP {
  MQCHAR4  StrucId;         /* Structure identifier */
  MQLONG   Version;         /* Structure version number */
  MQLONG   ExitOptions;     /* Reserved */
  MQLONG   AppOptions;      /* Application options */
  MQLONG   Encoding;        /* Encoding required by application */
  MQLONG   CodedCharSetId;  /* Coded character-set identifier required
                               by application */
  MQLONG   DataLength;      /* Length in bytes of message data */
  MQLONG   CompCode;        /* Completion code */
  MQLONG   Reason;          /* Reason code qualifying CompCode */
  MQLONG   ExitResponse;    /* Response from exit */
  MQHCONN  Hconn;           /* Connection handle */
 } MQDXP;
```

COBOL language declaration

```
**   MQDXP structure
  10 MQDXP.
**    Structure identifier
   15 MQDXP-STRUCID        PIC X(4).
**    Structure version number
   15 MQDXP-VERSION        PIC S9(9) BINARY.
**    Reserved
   15 MQDXP-EXITOPTIONS    PIC S9(9) BINARY.
**    Application options
   15 MQDXP-APPOPTIONS     PIC S9(9) BINARY.
**    Encoding required by application
   15 MQDXP-ENCODING       PIC S9(9) BINARY.
**    Coded character-set identifier required by application
```

```
   15 MQDXP-CODEDCHARSETID PIC S9(9) BINARY.
**    Length in bytes of message data
   15 MQDXP-DATALENGTH     PIC S9(9) BINARY.
**    Completion code
   15 MQDXP-COMPCODE       PIC S9(9) BINARY.
**    Reason code qualifying CompCode
   15 MQDXP-REASON         PIC S9(9) BINARY.
**    Response from exit
   15 MQDXP-EXITRESPONSE   PIC S9(9) BINARY.
**    Connection handle
   15 MQDXP-HCONN          PIC S9(9) BINARY.
```

System/390 assembler-language declaration (MVS/ESA only)

```
MQDXP                    DSECT
MQDXP_STRUCID            DS   CL4      Structure identifier
MQDXP_VERSION            DS   F        Structure version number
MQDXP_EXITOPTIONS        DS   F        Reserved
MQDXP_APPOPTIONS         DS   F        Application options
MQDXP_ENCODING           DS   F        Encoding required by
*                                      application
MQDXP_CODEDCHARSETID     DS   F        Coded character-set
*                                      identifier required by
*                                      application
MQDXP_DATALENGTH         DS   F        Length in bytes of message
*                                      data
MQDXP_COMPCODE           DS   F        Completion code
MQDXP_REASON             DS   F        Reason code qualifying
*                                      CompCode
MQDXP_EXITRESPONSE       DS   F        Response from exit
MQDXP_HCONN              DS   F        Connection handle
MQDXP_LENGTH             EQU  *-MQDXP  Length of structure
                         ORG  MQDXP
MQDXP_AREA               DS   CL(MQDXP_LENGTH)
```

MQXCNVC – Convert characters

The MQXCNVC call converts characters from one character set to another.

This call is part of the MQSeries Data Conversion Interface (DCI), which is one of the MQSeries framework interfaces. Note: this call can be used only from a data-conversion exit.

```
MQXCNVC (Hconn, Options, SourceCCSID, SourceLength, SourceBuffer,
         TargetCCSID, TargetLength, TargetBuffer, DataLength,
         CompCode, Reason)
```

Parameters

Hconn (MQHCONN) – input
Connection handle.

This handle represents the connection to the queue manager. It should normally be the handle passed to the data-conversion exit in the *Hconn* field of the MQDXP structure; this handle is not necessarily the same as the handle specified by the application which issued the MQGET call.

On OS/400, the following special value can be specified for *Hconn*:

MQHC_DEF_HCONN
Default connection handle.

Options (MQLONG) – input
Options that control the action of MQXCNVC.

Zero or more of the options described below can be specified. If more than one is required, the values can be:

- added together (do not add the same constant more than once), or
- combined using the bitwise OR operation (if the programming language supports bit operations).

Default-conversion option: The following option controls the use of default character conversion:

MQDCC_DEFAULT_CONVERSION
Default conversion.

This option specifies that default character conversion can be used if one or both of the character sets specified on the call is not supported. This allows the queue manager to use an installation-specified default character set that approximates the actual character set, when converting the string.

Note: The result of using an approximate character set to convert the string is that some characters may be converted incorrectly. This can be avoided by using in the string only characters which are common to both the actual character set specified on the call, and the default character set.

The default character set is specified by means of a configuration option when the queue manager is installed or restarted.

If this option is not specified, the queue manager uses only the specified character sets to convert the string, and the call fails if one or both of the character sets is not supported.

This option is supported in the following environments: AIX, HP-UX, OS/2, OS/400, Sun Solaris, Windows NT.

Encoding options: The options described below can be used to specify the integer encodings of the source and target strings. The relevant encoding is used *only* when the corresponding character set identifier indicates that the representation of the character set in main storage is dependent on the encoding used for binary integers. This affects only certain multibyte character sets (for example, UCS2 character sets).

The encoding is ignored if the character set is a single-byte character set (SBCS), or a multibyte character set whose representation in main storage is not dependent on the integer encoding.

These encoding options are supported in the following environments: AIX, HP-UX, OS/2, Sun Solaris, Windows NT.

Only one of the MQDCC_SOURCE_★ values should be specified, combined with one of the MQDCC_TARGET_★ values:

MQDCC_SOURCE_ENC_NATIVE
: Source encoding is the default for the environment and programming language.

MQDCC_SOURCE_ENC_NORMAL
: Source encoding is normal.

MQDCC_SOURCE_ENC_REVERSED
: Source encoding is reversed.

MQDCC_SOURCE_ENC_UNDEFINED
: Source encoding is undefined.

MQDCC_TARGET_ENC_NATIVE
: Target encoding is the default for the environment and programming language.

MQDCC_TARGET_ENC_NORMAL
: Target encoding is normal.

MQDCC_TARGET_ENC_REVERSED
: Target encoding is reversed.

MQDCC_TARGET_ENC_UNDEFINED
: Target encoding is undefined.

The encoding values defined above can be added directly to the *Options* field. However, if the source or target encoding is obtained from the *Encoding* field in the MQMD or other structure, the following processing must be done:

1. The integer encoding must be extracted from the *Encoding* field by eliminating the float and packed-decimal encodings; see “Analyzing encodings” on page 487 for details of how to do this.

2. The integer encoding resulting from step 1 must be multiplied by the appropriate factor before being added to the *Options* field. These factors are:

   MQDCC_SOURCE_ENC_FACTOR
   Factor for source encoding

   MQDCC_TARGET_ENC_FACTOR
   Factor for target encoding

The following illustrates how this might be coded in the C programming language:

```
Options = (MsgDesc.Encoding & MQENC_INTEGER_MASK)
           * MQDCC_SOURCE_ENC_FACTOR
         + (DataConvExitParms.Encoding & MQENC_INTEGER_MASK)
           * MQDCC_TARGET_ENC_FACTOR;
```

If not specified, the encoding options default to undefined (MQDCC_*_ENC_UNDEFINED). In most cases, this does not affect the successful completion of the MQXCNVC call. However, if the corresponding character set is a multibyte character set whose representation is dependent on the encoding (for example, a UCS2 character set), the call fails with reason code MQRC_SOURCE_INTEGER_ENC_ERROR or MQRC_TARGET_INTEGER_ENC_ERROR as appropriate.

Default option: If none of the options described above is specified, the following option can be used:

MQDCC_NONE
No options specified.

MQDCC_NONE is defined to aid program documentation. It is not intended that this option be used with any other, but as its value is zero, such use cannot be detected.

SourceCCSID (MQLONG) – input
Coded character set identifier of string before conversion.

This is the coded character set identifier of the input string in *SourceBuffer*.

SourceLength (MQLONG) – input
Length of string before conversion.

This is the length in bytes of the input string in *SourceBuffer*; it must be zero or greater.

SourceBuffer (MQCHAR×*SourceLength*) – input
String to be converted.

This is the buffer containing the string to be converted from one character set to another.

TargetCCSID (MQLONG) – input
Coded character set identifier of string after conversion.

This is the coded character set identifier of the character set to which *SourceBuffer* is to be converted.

TargetLength (MQLONG) – input
Length of output buffer.

This is the length in bytes of the output buffer *TargetBuffer*; it must be zero or greater.

TargetBuffer (MQCHAR×*TargetLength*) – output
String after conversion.

This is the string after it has been converted to the character set defined by *TargetCCSID*. The converted string can be shorter or longer than the unconverted string.

If *TargetBuffer* is too small to accommodate the converted string, the string is truncated to fit and the call completes with MQCC_WARNING and reason code MQRC_CONVERTED_MSG_TOO_BIG. The string is truncated in a way that ensures it remains a valid SBCS, DBCS, or mixed SBCS/DBCS string; this may result in the number of valid bytes returned in *TargetBuffer* being less than *TargetLength*. The *DataLength* parameter indicates the number of valid bytes returned.

DataLength (MQLONG) – output
Length of output string.

This is the length of the string returned in the output buffer *TargetBuffer*. The converted string can be shorter or longer than the unconverted string.

CompCode (MQLONG) – output
Completion code.

It is one of the following:

MQCC_OK
Successful completion.
MQCC_WARNING
Warning (partial completion).
MQCC_FAILED
Call failed.

Reason (MQLONG) – output
Reason code qualifying *CompCode*.

If *CompCode* is MQCC_OK:

MQRC_NONE
(0, X'000') No reason to report.

If *CompCode* is MQCC_WARNING:

MQRC_CONVERTED_MSG_TOO_BIG
(2120, X'848') Converted message too big for application buffer.

If *CompCode* is MQCC_FAILED:

MQRC_DATA_LENGTH_ERROR
(2010, X'7DA') Data length parameter not valid.
MQRC_DBCS_ERROR
(2150, X'866') DBCS string not valid.
MQRC_HCONN_ERROR
(2018, X'7E2') Connection handle not valid.

MQRC_OPTIONS_ERROR
(2046, X'7FE') Options not valid or not consistent.
MQRC_RESOURCE_PROBLEM
(2102, X'836') Insufficient system resources available.
MQRC_SOURCE_BUFFER_ERROR
(2145, X'861') Source buffer parameter not valid.
MQRC_SOURCE_CCSID_ERROR
(2111, X'83F') Source coded character set identifier not valid.
MQRC_SOURCE_INTEGER_ENC_ERROR
(2112, X'840') Source integer encoding not recognized.
MQRC_SOURCE_LENGTH_ERROR
(2143, X'85F') Source length parameter not valid.
MQRC_STORAGE_NOT_AVAILABLE
(2071, X'817') Insufficient storage available.
MQRC_TARGET_BUFFER_ERROR
(2146, X'862') Target buffer parameter not valid.
MQRC_TARGET_CCSID_ERROR
(2115, X'843') Target coded character set identifier not valid.
MQRC_TARGET_INTEGER_ENC_ERROR
(2116, X'844') Target integer encoding not recognized.
MQRC_TARGET_LENGTH_ERROR
(2144, X'860') Target length parameter not valid.
MQRC_UNEXPECTED_ERROR
(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see Chapter 5, “Return codes” on page 383.

C language invocation

```
MQXCNVC (Hconn, Options, SourceCCSID, SourceLength, SourceBuffer,
        TargetCCSID, TargetLength, TargetBuffer, &DataLength,
        &CompCode, &Reason);
```

Declare the parameters as follows:

```
MQHCONN  Hconn;            /* Connection handle */
MQLONG   Options;          /* Options that control the action of
                              MQXCNVC */
MQLONG   SourceCCSID;      /* Coded character set identifier of string
                              before conversion */
MQLONG   SourceLength;     /* Length of string before conversion */
MQCHAR   SourceBuffer[n];  /* String to be converted */
MQLONG   TargetCCSID;      /* Coded character set identifier of string
                              after conversion */
MQLONG   TargetLength;     /* Length of output buffer */
MQCHAR   TargetBuffer[n];  /* String after conversion */
MQLONG   DataLength;       /* Length of output string */
MQLONG   CompCode;         /* Completion code */
MQLONG   Reason;           /* Reason code qualifying CompCode */
```

System/390 assembler-language invocation (MVS/ESA only)

```
          CALL MQXCNVC,(HCONN,OPTIONS,SOURCECCSID,SOURCELENGTH,           X
                SOURCEBUFFER,TARGETCCSID,TARGETLENGTH,TARGETBUFFER,       X
                DATALENGTH,COMPCODE,REASON)
```

Declare the parameters as follows:

```
HCONN               DS      F           Connection handle
OPTIONS             DS      F           Options that control the action
*                                       of MQXCNVC
SOURCECCSID         DS      F           Coded character set identifier
*                                       of string before conversion
SOURCELENGTH        DS      F           Length of string before
*                                       conversion
SOURCEBUFFER        DS      CL(n)       String to be converted
TARGETCCSID         DS      F           Coded character set identifier
*                                       of string after conversion
TARGETLENGTH        DS      F           Length of output buffer
TARGETBUFFER        DS      CL(n)       String after conversion
DATALENGTH          DS      F           Length of output string
COMPCODE            DS      F           Completion code
REASON              DS      F           Reason code qualifying CompCode
```

MQDATACONVEXIT – Data conversion exit

This call definition describes the parameters that are passed to the data-conversion exit. No entry point called MQDATACONVEXIT is actually provided by the queue manager (see usage note 11 on page 518).

This definition is part of the MQSeries Data Conversion Interface (DCI), which is one of the MQSeries framework interfaces.

```
MQDATACONVEXIT (DataConvExitParms, MsgDesc, InBufferLength,
                InBuffer, OutBufferLength, OutBuffer)
```

Parameters

DataConvExitParms (MQDXP) – input/output
Data-conversion exit parameter block.

This structure contains information relating to the invocation of the exit. The exit sets information in this structure to indicate the outcome of the conversion. See "MQDXP – Data-conversion exit parameter structure" on page 502 for details of the fields in this structure.

MsgDesc (MQMD) – input/output
Message descriptor.

On input to the exit, this is the message descriptor that would be returned to the application if no conversion were performed. It therefore contains the *Format*, *Encoding*, and *CodedCharSetId* of the unconverted message contained in *InBuffer*.

Note: The *MsgDesc* parameter passed to the exit is always the most-recent version of MQMD supported by the queue manager which invokes the exit. If the exit is intended to be portable between different environments, the exit should check the *Version* field in *MsgDesc* to verify that the fields that the exit needs to access are present in the structure.

In the following environments, the exit is passed a version-2 MQMD: AIX, HP-UX, OS/2, OS/400, Sun Solaris, Windows NT. In all other environments that support the data conversion exit, the exit is passed a version-1 MQMD.

On output, the exit should change the *Encoding* and *CodedCharSetId* fields to the values requested by the application, if conversion was successful; these changes will be reflected back to the application. Any other changes that the exit makes to the structure are ignored; they are not reflected back to the application.

InBufferLength (MQLONG) – input
Length in bytes of *InBuffer*.

This is the length of the input buffer *InBuffer*, and specifies the number of bytes to be processed by the exit. *InBufferLength* is the lesser of the length of the message data prior to conversion, and the length of the buffer provided by the application on the MQGET call.

The value is always greater than zero.

InBuffer (MQBYTE×*InBufferLength*) – input
Buffer containing the unconverted message.

This contains the message data prior to conversion. If the exit is unable to convert the data, the queue manager returns the contents of this buffer to the application after the exit has completed.

Note: The exit should not alter *InBuffer*; if this parameter is altered, the results are undefined.

In the C programming language, this parameter is defined as a pointer-to-void.

OutBufferLength (MQLONG) – input
Length in bytes of *OutBuffer*.

This is the length of the output buffer *OutBuffer*, and is the same as the length of the buffer provided by the application on the MQGET call.

The value is always greater than zero.

OutBuffer (MQBYTE×*OutBufferLength*) – output
Buffer containing the converted message.

On output from the exit, if the conversion was successful (as indicated by the value MQXDR_OK in the *ExitResponse* field of the *DataConvExitParms* parameter), *OutBuffer* contains the message data to be delivered to the application, in the requested representation. If the conversion was unsuccessful, any changes that the exit has made to this buffer are ignored.

In the C programming language, this parameter is defined as a pointer-to-void.

Usage notes

1. A data-conversion exit is a user-written exit which receives control during the processing of an MQGET call. The function performed by the data-conversion exit is defined by the provider of the exit; however, the exit must conform to the rules described here, and in the associated parameter structure MQDXP.

   The programming languages that can be used for a data-conversion exit are determined by the environment.

2. The exit is invoked only if *all* of the following are true:

   - The MQGMO_CONVERT option is specified on the MQGET call
   - The *Format* field in the message descriptor is not MQFMT_NONE
   - The message is not already in the required representation; that is, one or both of the message's *CodedCharSetId* and *Encoding* is different from the value specified by the application in the message descriptor supplied on the MQGET call
   - The queue manager has not already done the conversion successfully
   - The length of the application's buffer is greater than zero
   - The length of the message data is greater than zero

- The reason code so far during the MQGET operation is MQRC_NONE or MQRC_TRUNCATED_MSG_ACCEPTED

3. When an exit is being written, consideration should be given to coding the exit in a way that will allow it to convert messages that have been truncated. Truncated messages can arise in the following ways:

   - The receiving application provides a buffer that is smaller than the message, but specifies the MQGMO_ACCEPT_TRUNCATED_MSG option on the MQGET call.

     In this case, the *Reason* field in the *DataConvExitParms* parameter on input to the exit will have the value MQRC_TRUNCATED_MSG_ACCEPTED.

   - The sender of the message truncated it before sending it. This can happen with report messages, for example (see "Conversion of report messages" on page 501 for more details).

     In this case, the *Reason* field in the *DataConvExitParms* parameter on input to the exit will have the value MQRC_NONE (if the receiving application provided a buffer that was big enough for the message).

   Thus the value of the *Reason* field on input to the exit cannot always be used to decide whether the message has been truncated.

   The distinguishing characteristic of a truncated message is that the length provided to the exit in the *InBufferLength* parameter will be *less than* the length implied by the format name contained in the *Format* field in the message descriptor. The exit should therefore check the value of *InBufferLength* before attempting to convert any of the data; the exit *should not* assume that the full amount of data implied by the format name has been provided.

   If the exit has *not* been written to convert truncated messages, and *InBufferLength* is less than the value expected, the exit should return MQXDR_CONVERSION_FAILED in the *ExitResponse* field of the *DataConvExitParms* parameter, with the *CompCode* and *Reason* fields set to MQCC_WARNING and MQRC_FORMAT_ERROR respectively.

   If the exit *has* been written to convert truncated messages, the exit should convert as much of the data as possible (see next usage note), taking care not to attempt to examine or convert data beyond the end of *InBuffer*. If the conversion completes successfully, the exit should leave the *Reason* field in the *DataConvExitParms* parameter unchanged. This has the effect of returning MQRC_TRUNCATED_MSG_ACCEPTED if the message was truncated by the receiver's queue manager, and MQRC_NONE if the message was truncated by the sender of the message.

   It is also possible for a message to expand *during* conversion, to the point where it is bigger than *OutBuffer*. In this case the exit must decide whether to truncate the message; the *AppOptions* field in the *DataConvExitParms* parameter will indicate whether the receiving application specified the MQGMO_ACCEPT_TRUNCATED_MSG option.

4. Generally it is recommended that all of the data in the message provided to the exit in *InBuffer* is converted, or that none of it is. An exception to this, however, occurs if the message is truncated, either before conversion or during conversion; in this case there may be an incomplete item at the end of the buffer (for example: one byte of a double-byte character, or 3 bytes of a 4-byte integer). In this situation it is recommended that the incomplete item should be

omitted, and unused bytes in *OutBuffer* set to nulls. However, complete elements or characters within an array or string *should* be converted.

5. When an exit is needed for the first time, the queue manager attempts to load an object that has the same name as the format (apart from environment-specific extensions). The object loaded must contain the exit that processes messages with that format name. It is recommended that the exit name, and the name of the object that contain the exit, should be identical, although not all environments require this.
6. A new copy of the exit is loaded when an application attempts to retrieve the first message that uses that *Format* since the application connected to the queue manager. For CICS or IMS applications, this means when the CICS or IMS subsystem connected to the queue manager. A new copy may also be loaded at other times, if the queue manager has discarded a previously-loaded copy. For this reason, an exit should not attempt to use static storage to communicate information from one invocation of the exit to the next – the exit may be unloaded between the two invocations.
7. If there is a user-supplied exit with the same name as one of the built-in formats supported by the queue manager, the user-supplied exit does not replace the built-in conversion routine. The only circumstances in which such an exit is invoked are:
   - If the built-in conversion routine cannot handle conversions to or from either the *CodedCharSetId* or *Encoding* involved, or
   - If the built-in conversion routine has failed to convert the data (for example, because there is a field or character which cannot be converted).
8. The scope of the exit is environment-dependent. *Format* names should be chosen so as to minimize the risk of clashes with other formats. It is recommended that they start with characters that identify the application defining the format name.
9. The data-conversion exit runs in an environment similar to that of the program which issued the MQGET call; environment includes address space and user profile (where applicable). The program could be a message channel agent sending messages to a destination queue manager that does not support message conversion. The exit cannot compromise the queue manager's integrity, since it does not run in the queue manager's environment.
10. The only MQI call which can be used by the exit is MQXCNVC; attempting to use other MQI calls fails with reason code MQRC_CALL_IN_PROGRESS, or other unpredictable errors.
11. No entry point called MQDATACONVEXIT is actually provided by the queue manager. However, a **typedef** is provided for the name MQDATACONVEXIT in the C programming language, and this can be used to declare the user-written exit, to ensure that the parameters are correct. The name of the exit should be the same as the format name (the name contained in the *Format* field in MQMD), although this is not required in all environments.

    The following example illustrates how the exit that processes the format MYFORMAT should be declared in the C programming language:

```
#include "cmqc.h"
#include "cmqxc.h"

MQDATACONVEXIT MYFORMAT;

void MQENTRY MYFORMAT(
     PMQDXP   pDataConvExitParms, /* Data-conversion exit parameter
                                     block */
     PMQMD    pMsgDesc,           /* Message descriptor */
     MQLONG   InBufferLength,     /* Length in bytes of InBuffer */
     PMQVOID  pInBuffer,          /* Buffer containing the unconverted
                                     message */
     MQLONG   OutBufferLength,    /* Length in bytes of OutBuffer */
     PMQVOID  pOutBuffer)         /* Buffer containing the converted
                                     message */
{
  /* C language statements to convert message */
}
```

12. On MVS/ESA, if an API-crossing exit is also in force, it is called after the data-conversion exit.

C invocation

```
exitname (&DataConvExitParms, &MsgDesc, InBufferLength,
         InBuffer, OutBufferLength, OutBuffer);
```

Declare the parameters as follows:

```
MQDXP   DataConvExitParms;  /* Data-conversion exit parameter block */
MQMD    MsgDesc;            /* Message descriptor */
MQLONG  InBufferLength;     /* Length in bytes of InBuffer */
MQBYTE  InBuffer[n];        /* Buffer containing the uncon-
                               verted message */
MQLONG  OutBufferLength;    /* Length in bytes of OutBuffer */
MQBYTE  OutBuffer[n];       /* Buffer containing the converted
                               message */
```

COBOL invocation (OS/400 only)

```
     CALL 'exitname' USING DATACONVEXITPARMS, MSGDESC,
                           INBUFFERLENGTH, INBUFFER, OUTBUFFERLENGTH,
                           OUTBUFFER.
```

Declare the parameters as follows:

```
**   Data-conversion exit parameter block
 01  DATACONVEXITPARMS.
     COPY CMQDXPV.
**   Message descriptor
 01  MSGDESC.
     COPY CMQMDV.
**   Length in bytes of InBuffer
 01  INBUFFERLENGTH     PIC S9(9) BINARY.
**   Buffer containing the unconverted message
 01  INBUFFER           PIC X(n).
**   Length in bytes of OutBuffer
 01  OUTBUFFERLENGTH    PIC S9(9) BINARY.
**   Buffer containing the converted message
 01  OUTBUFFER          PIC X(n).
```

System/390 assembler-language invocation (MVS/ESA only)

```
          CALL EXITNAME,(DATACONVEXITPARMS,MSGDESC,INBUFFERLENGTH,INBUFFER, X
                OUTBUFFERLENGTH,OUTBUFFER)
```

Declare the parameters as follows:

```
DATACONVEXITPARMS   CMQDXPA             Data conversion exit parameter
*                                       block
MSGDESC             CMQMDA              Message descriptor
INBUFFERLENGTH      DS      F           Length in bytes of InBuffer
INBUFFER            DS      CL(n)       Buffer containing the
*                                       unconverted message
OUTBUFFERLENGTH     DS      F           Length in bytes of OutBuffer
OUTBUFFER           DS      CL(n)       Buffer containing the converted
*                                       message
```

End of Product-sensitive programming interface

Appendix E. Signal notification IPC message (Tandem NSK only)

For backwards compatibility with MQSeries for Tandem NSK, Version 1.5.1, the signal mode of message-arrival notification is supported. This type of notification is selected by the MQGMO_SET_SIGNAL option in the options field of the Get Message Options structure. If MQGMO_SET_SIGNAL is specified, the following options are not valid:

- MQGMO_BROWSE_FIRST
- MQGMO_BROWSE_NEXT
- MQGMO_BROWSE_MSG_UNDER_CURSOR
- MQGMO_MSG_UNDER_CURSOR
- MQGMO_LOCK
- MQGMO_UNLOCK
- MQGMO_WAIT

If MQGMO_SET_SIGNAL is specified with any of these options, a *CompCode* of MQCC_FAILED and a *Reason* of MQRC_OPTIONS_ERROR are returned.

The effects of specifying MQGMO_SET_SIGNAL are as follows:

- If a message is available when MQGET is issued, it is returned immediately to the requesting application.
- If no message is available when MQGET is issued, a *CompCode* of MQCC_WARNING and a *Reason* of MQRC_SIGNAL_REQUEST_ACCEPTED are returned. When a message becomes available, an Inter-Process Communication (IPC) message is sent to the $RECEIVE queue of the process that made the MQGET call.

The format of this IPC message is:

MsgCode (INT)
Identifies the message as a notification. The value is TRIGGER_RESPONSE.

ApplTag (LONG)
Is the application tag provided in the *Signal1* field of MQGMO.

The *Signal1* field of MQGMO is significant only when the signal mode of message-arrival notification has been requested. It can be used by an application to associate the IPC notification message with a particular MQGET request.

Status (LONG)
Is the reason Code from MQGET. It can have the following values:

MQRC_NONE
A message satisfying the criteria specified in the MQGET call is available on the queue.

MQRC_NO_MSG_AVAILABLE
The time specified in the *WaitInterval* field has expired.

MQRC_CONNECTION_BROKEN
The queue manager has been stopped.

MQRC_GET_INHIBITED
An operator has inhibited the GET operation for the queue.

MQRC_Q_DELETED
The queue has been deleted.

MQRC_Q_MGR_QUIESCING
The queue manager is quiescing, and the MQGET call was issued with the MQGMO_FAIL_IF_QUIESCING option.

MQRC_Q_MGR_STOPPING
The queue manager is shutting down.

Only one signal-notification-mode MQGET call can be outstanding for any queue. If an MQGET with signal notification is specified when there is already a signal-notification MQGET call outstanding for the same queue, a *CompCode* of MQCC_FAILED and a *Reason* of MQRC_SIGNAL_OUSTANDING are returned.

If the signal notification indicates that a message is available (*Status* is MQRC_NONE), the message is not locked by the Queue Manager; therefore, it is also available to any other application that shares the queue. It is possible, therefore, that the message will not be available by the time the application issues an MQGET call to retrieve or browse the message. The signal notification IPC message is not part of any unit of work (that is, a Tandem TMF transaction), started by either the application or MQSeries.

If the application calls MQCLOSE for a queue with outstanding signal-notification MQGET operations initiated by that application, the outstanding signal notifications are cancelled. If an application calls MQDISC, all outstanding signal notifications initiated by the application are cancelled.

Appendix F. Code page conversion tables

Each of the tables shows the conversion support for the characters used by one language.

Some of the coded character set identifiers (CCSIDs) are used by many languages, for example CCSID 819 (ISO8859-1 Western European), and appear in many tables. Other CCSIDs, for example CCSID 273 (German EBCDIC), appear in only one table.

The following terms are used in the tables:

ISO	Indicates that the CCSID is for an ISO 8859 codeset
pc-A	Indicates in the AIX and GIS rows that the CCSID is an IBM defined CCSID used in AIX, AT&T, and OS/2.
-8	Indicates in the HP-UX rows that the CCSID is for the HP-UX defined codeset *roman8*
GIS	Indicates MQSeries for AT&T GIS UNIX
NT	Indicates MQSeries for Windows NT
Solaris	Indicates MQSeries for Sun Solaris
SunOS	Indicates MQSeries for SunOS
SINIX, DC/OSx	Indicates MQSeries for SINIX and DC/OSx
DEC-OVMS	Indicates MQSeries for Digital OpenVMS
Tandem	Indicates MQSeries for Tandem NonStop Kernel V2.2

The following codes are used in the tables:

Y	Conversion at target supported going to and from source
y	No conversion is required because the different MQSeries products are operating in the same CCSID

The default for data conversion is for the conversion to be performed at the target (receiving) system.

Where a cell in a table is blank, conversion is not supported by the target product.

If the source product supports the conversion a channel can be set up and data exchanged by setting the channel attribute **DataConversion** to YES at the source. To determine if the source product supports the conversion, read the relevant table with source and target reversed. If conversion is shown as supported, it is possible to do conversion in the source product.

Notes:

1. If you have MQSeries for MVS/ESA V1.1.3 and have installed APAR PN73611, you can change the default CCSID. If you have an earlier release, or have not applied this APAR, CCSID 500 is always used; this means that you can only use the multilingual code page (Table 78 on page 534).
2. Conversion for MQSeries client information takes place in the server, so the server must support conversion from the client CCSID to the server CCSID.

3. The OS/2 and Solaris rows include information from some country specific versions. Not all of the conversions shown in the OS/2 and Solaris rows are supported by all OS/2 and Solaris versions.

For an extended list of CCSIDs, see the *Character Data Representation Reference*. See Table 69 for a cross reference between some of the CCSID numbers and some industry codeset names.

Table 69. Codeset names and CCSIDs

Codeset names	CCSIDs
ISO 8859-1	819
ISO 8859-2	912
ISO 8859-5	915
ISO 8859-6	1089
ISO 8859-7	813
ISO 8859-8	916
ISO 8859-9	920
big5	950
eucJP	954 5050 33722
eucKR	970
eucTW	964
eucCN	1383

MQSeries for MVS/ESA V1.1.4 or later provides conversions between single byte CCSIDs in addition to those listed in the language tables. A complete list of conversions provided is shown in Table 100 on page 561.

MQSeries for OS2 Warp V5 provides conversions between CCSIDs in addition to those listed in the language tables. A complete list of conversions provided is shown in "OS/2 conversion support" on page 567.

Where OS/400 operating system levels are indicated these should be at the following PTF levels or later:

V3R2 SF43993
V3R6 SF43804
V3R7 SF38997
V4R1 SF44021
V4R2 SF43902

Code page conversion tables

The following tables show the conversion support, between the source and target systems, for each of the national languages.

Table 70. Conversion support: US ENGLISH

Target ▼ / Source →	MVS, OS/400	OS/2, GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	AIX, GIS, NT	HP-UX	Windows client	Apple client
CCSID	37	437	819	850	1051	1252	1275
MVS	37	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400	37	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	**Y§**
OS/2	437	**Y**	*y*	**Y††**	**Y**	**Y††**	**Y††**	**Y††**
AIX (pc-A)	850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO)	819	**Y**	**Y***	*y*	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8)	1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A)	437	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A)	850	**Y**	**Y**	**Y**	*y*	**Y**		
NT	437	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT	850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS	819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx	819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS	819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem	819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 71. Conversion support: GERMAN

Target ↓ / Source →	MVS, OS/400	GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	HP-UX	Windows client	Apple client
CCSID	273	437	819	850	1051	1252	1275
MVS 273	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 273	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2 850	**Y**	**Y**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y**	**Y***	*y*	**Y**	**Y***		
HP-UX (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A) 437	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y**		
NT 437	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx 819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS 819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem 819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 72. Conversion support: DANISH and NORWEGIAN

Target ▼ / Source →	MVS, OS/400	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	OS/2, GIS, NT	HP-UX	Windows client	Apple client
CCSID	277	819	850	865	1051	1252	1275
MVS 277	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 277	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2 850	**Y**	**Y††**	*y*	**Y**	**Y††**	**Y††**	**Y††**
OS/2 865	**Y**	**Y††**	**Y**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	*y*		**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y**	*y*	**Y**		**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**		*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A) 865	**Y**	**Y**	**Y**	*y*			
NT 850	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
NT 865	**Y**	**Y**	**Y**	*y*		**Y††**	
Solaris 819	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	*y*	**Y**				
SINIX, DC/OSx 819	**Y**	*y*	**Y**	**Y**	**Y**		
DEC-OVMS 819	**Y**	*y*	**Y**	**Y**	**Y**		
Tandem 819	**Y**	*y*	**Y**	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 73 (Page 1 of 2). Conversion support: FINNISH and SWEDISH

Target ↓ **Source** →	**MVS, OS/400**	**GIS, NT**	**AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem**	**OS/2, AIX, GIS, NT**	**OS/2, NT**	**HP-UX**	**Windows client**	**Apple client**
CCSID	278	437	819	850	865	1051	1252	1275
MVS 278	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 278	*y*	**Y**	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2 850	**Y**	**Y**	**Y††**	*y*	**Y**	**Y††**	**Y††**	**Y††**
OS/2 865	**Y**	**Y**	**Y††**	**Y**	**Y**	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	**Y**	*y*		**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y**	**Y***	*y*	**Y**		**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**	**Y**		*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A) 437	**Y**	*y*	**Y**	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y**		
NT 437	**Y**	*y*	**Y**	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
NT 865	**Y**	**Y**	**Y**	**Y**	*y*		**Y††**	
Solaris 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	**Y**	*y*	**Y**				
SINIX, DC/OSx 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		
DEC-OVMS 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		

Table 73 (Page 2 of 2). Conversion support: FINNISH and SWEDISH

Target ▼	Source →	MVS, OS/400	GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	OS/2, NT	HP-UX	Windows client	Apple client
Tandem	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 74. Conversion support: ITALIAN

Target ↓ / Source →	MVS, OS/400	GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	HP-UX	Windows client	Apple client
CCSID	280	437	819	850	1051	1252	1275
MVS 280	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 280	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2 850	**Y**	**Y**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y**	**Y***	*y*	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A) 437	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y**		
NT 437	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx 819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS 819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem 819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 75. Conversion support: SPANISH

Target ▼ / Source →	MVS, OS/400	GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	HP-UX	Windows client	Apple client
CCSID	284	437	819	850	1051	1252	1275
MVS 284	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 284	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2 850	**Y**	**Y**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y**	**Y***	*y*	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A) 437	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y**		
NT 437	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx 819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS 819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem 819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 76. Conversion support: UK ENGLISH / GAELIC

Target ↓	Source →	MVS, OS/400	GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	HP-UX	Windows client	Apple client
	CCSID	285	437	819	850	1051	1252	1275
MVS	285	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400	285	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2	850	**Y**	**Y**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A)	850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO)	819	**Y**	**Y***	*y*	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8)	1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A)	437	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A)	850	**Y**	**Y**	**Y**	*y*	**Y**		
NT	437	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT	850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS	819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx	819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS	819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem	819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 77. Conversion support: FRENCH

Target ▼ / Source →	MVS, OS/400	GIS, NT	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	HP-UX	Windows client	Apple client
CCSID	297	437	819	850	1051	1252	1275
MVS 297	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 297	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	
OS/2 850	**Y**	**Y**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y**	**Y***	*y*	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A) 437	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y**		
NT 437	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx 819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS 819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem 819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 78. Conversion support: MULTILINGUAL

Target ↓ / Source →	GIS, NT	MVS, OS/400	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	HP-UX	Windows client	Apple client
CCSID	437	500	819	850	1051	1252	1275
MVS 500	**Y†**	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400 500	**Y**	*y*	**Y**	**Y**	**Y§**	**Y**	**Y§**
OS/2 850	**Y**	**Y**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y***	**Y††**	**Y††**
AIX (ISO) 819	**Y***	**Y**	*y*	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**	**Y**	**Y**	*y*	**Y††**	**Y††**
GIS (ISO) 819	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A) 437	*y*	**Y**	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	**Y**	**Y**	*y*	**Y**		
NT 437	*y*	**Y**	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 850	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris 819	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	**Y**	**Y**	*y*	**Y**			
SINIX, DC/OSx 819	**Y**	**Y**	*y*	**Y**	**Y**		
DEC-OVMS 819	**Y**	**Y**	*y*	**Y**	**Y**		
Tandem 819	**Y**	**Y**	*y*	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 79 (Page 1 of 2). Conversion support: PORTUGUESE

Target ▼	**Source** →	**OS/400**	**MVS, OS/400**	**AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem**	**OS/2, AIX, GIS, NT**	**OS/2, GIS, NT**	**HP-UX**	**Windows client**	**Apple client**
	CCSID	37	500	819	850	860	1051	1252	1275
MVS	500	**Y†**	*y*	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**	**Y†**
OS/400	37	*y*	**Y**	**Y**	**Y**	**Y**	**Y§**	**Y**	**Y§**
OS/400	500	**Y**	*y*	**Y**	**Y**	**Y**	**Y§**	**Y**	**Y§**
OS/2	850	**Y**	**Y**	**Y††**	*y*	**Y**	**Y††**	**Y††**	**Y††**
OS/2	860	**Y**	**Y**	**Y††**	**Y**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A)	850	**Y**	**Y**	**Y**	*y*	**Y††**	**Y***	**Y††**	**Y††**
AIX (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y††**	**Y***	**Y††**	**Y††**
HP-UX (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8)	1051	**Y**	**Y**	**Y**	**Y**		*y*	**Y††**	**Y††**
GIS (ISO)	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		
GIS (pc-A)	850	**Y**	**Y**	**Y**	*y*	**Y**	**Y**		
GIS (pc-A)	860	**Y**	**Y**	**Y**	**Y**	*y*			
NT	850	**Y**	**Y**	**Y**	*y*	**Y**	**Y**	**Y††**	**Y††**
NT	860	**Y**	**Y**	**Y**	**Y**	*y*		**Y††**	
Solaris	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
SunOS	819	**Y**	**Y**	*y*	**Y**				
SINIX, DC/OSx	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		
DEC-OVMS	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		

Table 79 (Page 2 of 2). Conversion support: PORTUGUESE

Target ↓	Source→	OS/400	MVS, OS/400	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, AIX, GIS, NT	OS/2, GIS, NT	HP-UX	Windows client	Apple client
Tandem	819	**Y**	**Y**	*y*	**Y**	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 80. Conversion support: ICELANDIC

Target ▼ / Source →	AIX, HP-UX, GIS, Solaris, SunOS, DEC-OVMS, Tandem	OS/2, GIS, NT	OS/2, AIX, NT	MVS, OS/400	HP-UX	Windows client	Apple client
CCSID	819	850	861	871	1051	1252	1275
MVS 871	**Y†**	**Y†**	**Y†**	*y*	**Y†**	**Y†**	**Y†**
OS/400 871	**Y**	**Y**	**Y**	*y*	**Y§**	**Y**	
OS/2 850	**Y††**	*y*	**Y**	**Y**	**Y††**	**Y††**	**Y††**
OS/2 861	**Y††**	**Y**	*y*	**Y**	**Y††**	**Y††**	**Y††**
AIX (pc-A) 850	**Y**	*y*	**Y††**	**Y**	**Y***	**Y††**	**Y††**
AIX (ISO) 819	*y*	**Y**	**Y††**	**Y**	**Y***	**Y††**	**Y††**
HP-UX (ISO) 819	*y*	**Y**	**Y**	**Y**	**Y**	**Y††**	**Y††**
HP-UX (-8) 1051	**Y**	**Y**		**Y**	*y*	**Y††**	**Y††**
GIS (ISO) 819	*y*	**Y**	**Y**	**Y**	**Y**		
GIS (pc-A) 850	**Y**	*y*	**Y**	**Y**	**Y**		
NT 850	**Y**	*y*	**Y**	**Y**	**Y**	**Y††**	**Y††**
NT 861	**Y**	**Y**	*y*	**Y**		**Y††**	
Solaris 819	*y*	**Y**	**Y**	**Y**	**Y**	**Y††**	**Y††**
SunOS 819	*y*	**Y**		**Y**			
SINIX, DC/OSx 819	*y*	**Y**	**Y**	**Y**	**Y**		
DEC-OVMS 819	*y*	**Y**	**Y**	**Y**	**Y**		
Tandem 819	*y*	**Y**	**Y**	**Y**	**Y**		

Note:

***** Supported on MQSeries for AIX version 2.2.1 or later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 81. Conversion support: EASTERN EUROPEAN Languages

Target ↓	Source →	OS/2, NT	MVS, OS/400	AIX, HP-UX, GIS, Solaris, DEC-OVMS, Tandem	Windows client	Apple client Eastern European	Apple client Croatian	Apple client Romanian
	CCSID	852	870	912	1250	1282	1284	1285
MVS	870	**Y†**	*y*	**Y†**	**Y†**	**Y†**		
OS/400	870	**Y**	*y*	**Y**	**Y**	**Y§**		
OS/2	852	*y*	**Y**	**Y††**	**Y††**	**Y††**	**Y††**	**Y††**
AIX (ISO)	912	**Y***	**Y***	*y*	**Y††**	**Y††**	**Y††**	**Y††**
HP-UX (ISO)	912	**Y**	**Y**	*y*	**Y††**	**Y††**		
GIS (ISO)	912	**Y**	**Y**	*y*				
NT	852	*y*	**Y**	**Y**	**Y††**	**Y††**		
Solaris	912	**Y**	**Y**	*y*	**Y††**	**Y††**		
SunOS								
SINIX, DC/OSx	912	**Y**	**Y**	*y*				
DEC-OVMS	912	**Y**	**Y**	*y*				
Tandem	912	**Y**	**Y**	*y*				

Note: The typical languages which use these CCSIDS include Albanian, Croatian, Czech, Hungarian, Polish, Romanian, Serbian, Slovakian, and Sloven.

Note:

***** Only on AIX V4.1 and later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 82 (Page 1 of 2). Conversion support: CYRILLIC

Target ▼	**Source** →	**OS/2, NT**	**OS/2, NT**	**OS/400**	**AIX, HP-UX, GIS, Solaris, DEC-OVMS, Tandem**	**MVS, OS/400**	**OS/2, NT**	**Windows client**	**Apple client**
	CCSID	855	866	880	915	1025	1131	1251	1283
MVS	1025	**Y†**	**Y†**	**Y†**	**Y†**	*y*		**Y†**	**Y†**
OS/400	880			*y*	**Y**	**Y**		**Y**	**Y§**
OS/400	1025	**Y**	**Y**	**Y**	**Y**	*y*	**Y§**	**Y**	**Y§**
OS/2	855	*y*	**Y**	**Y††**	**Y**	**Y**	**Y††**	**Y††**	**Y††**
OS/2	866	**Y**	*y*	**Y††**	**Y**	**Y**	**Y††**	**Y††**	**Y††**
OS/2	1131	**Y††**	**Y††**	**Y††**	**Y††**	**Y††**	*y*	**Y††**	**Y††**
AIX (ISO)	915	**Y+**	**Y***	**Y+**	*y*	**Y+**	**Y††**	**Y††**	**Y††**
HP-UX (ISO)	915	**Y**	**Y****	**Y**	*y*	**Y**	**Y††**	**Y††**	**Y††**
GIS (ISO)	915	**Y**	**Y**	**Y**	*y*	**Y**			
NT	855	*y*	**Y**	**Y**	**Y**	**Y**		**Y††**	**Y††**
NT	866	**Y**	*y*	**Y**	**Y**	**Y**		**Y††**	**Y††**
NT	1131				**Y††**	**Y††**	*y*	**Y††**	
Solaris	915	**Y**	**Y**	**Y**	*y*	**Y**	**Y††**	**Y††**	**Y††**
SunOS									
SINIX, DC/OSx	915	**Y**	**Y**	**Y**	*y*	**Y**			
DEC-OVMS	915	**Y**	**Y**	**Y**	*y*	**Y**			

Table 82 (Page 2 of 2). Conversion support: CYRILLIC

Target ↓ / Source →	OS/2, NT	OS/2, NT	OS/400	AIX, HP-UX, GIS, Solaris, DEC-OVMS, Tandem	MVS, OS/400	OS/2, NT	Windows client	Apple client
Tandem	915	**Y**	**Y**	**Y**	*y*	**Y**			

Note: The typical languages which use these CCSIDS include Byelorussia (Belarus), Bulgarian, Macedonian, Russian, and Serbian.

Note:

+ Only on AIX V4.1 and later.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
***** Supported on MQSeries for AIX version 2.2.1 or later.
****** Supported on MQSeries for HP version 2.2.1 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 83. Conversion support: ESTONIAN

Target ↓	**Source** →	**OS/2, AIX, HP-UX, NT, Solaris**	**MVS, OS/400**	**Windows client**
	CCSID	922	1122	1257
MVS	1122	**Y†**	*y*	**Y†**
OS/400	1122	**Y§**	*y*	**Y**
AIX	922	*y*	**Y††**	**Y††**
OS/2	922	*y*	**Y††**	**Y††**
HP-UX	922	*y*	**Y††**	**Y††**
GIS (ISO)				
NT	922	*y*	**Y††**	**Y††**
Solaris	922	*y*	**Y††**	**Y††**
SunOS				
SINIX, DC/OSx				
DEC-OVMS				
Tandem				

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R6, V3R7, V4R1 or later.

Table 84. Conversion support: LATVIAN and LITHUANIAN

Target ↓	**Source** →	**OS/2, AIX, HP-UX, NT, Solaris**	**MVS, OS/400**	**Windows client**
	CCSID	921	1112	1257
MVS	1112	**Y†**	*y*	**Y†**
OS/400	1112	**Y§**	*y*	**Y**
OS/2	921	*y*	**Y††**	**Y††**
AIX	921	*y*	**Y††**	**Y††**
HP-UX	921	*y*	**Y††**	**Y††**
GIS (ISO)				
NT	921	*y*	**Y††**	**Y††**
Solaris	921	*y*	**Y††**	**Y††**
SunOS				
SINIX, DC/OSx				
DEC-OVMS				
Tandem				

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R6, V3R7, V4R1 or later.

Table 85. Conversion support: UKRAINIAN

Target ▼ / Source ►	MVS, OS/400	AIX, HP-UX, NT, Solaris	OS/2, NT	Windows client
CCSID	1123	1124	1125	1251
MVS 1123	*y*	**Y†**		**Y†**
OS/400 1123	*y*		**Y§**	
OS/2 1125	**Y††**	**Y††**	*y*	**Y††**
AIX 1124	**Y††**	*y*	**Y††**	**Y††**
HP-UX 1124	**Y††**	*y*	**Y††**	**Y††**
GIS (ISO)				
NT 1124	**Y††**	*y*	**Y††**	**Y††**
NT 1125		**Y††**	*y*	
Solaris 1124	**Y††**	*y*	**Y††**	**Y††**
SunOS				
SINIX, DC/OSx				
DEC-OVMS				
Tandem				

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 86. Conversion support: GREEK

Target ↓	Source →	OS/2, AIX, HP-UX, GIS, Solaris, DEC-OVMS, Tandem	OS/2, NT	MVS, OS/400	Windows client	Apple client
	CCSID	813	869	875	1253	1280
MVS	875	**Y†**	**Y†**	*y*	**Y†**	**Y†**
OS/400	875	**Y**	**Y**	*y*	**Y**	**Y§**
OS/2	813	*y*	**Y**	**Y**	**Y††**	**Y††**
OS/2	869	**Y**	*y*	**Y**	**Y††**	**Y††**
AIX (ISO)	813	*y*	**Y**	**Y**	**Y††**	**Y††**
HP-UX (ISO)	813#	*y*	**Y**	**Y**	**Y††**	**Y††**
GIS (ISO)	813	*y*	**Y**	**Y**		
NT	869	**Y**	*y*	**Y**	**Y††**	**Y††**
Solaris	813	*y*	**Y**	**Y**	**Y††**	**Y††**
SunOS						
SINIX, DC/OSx	813	*y*	**Y**	**Y**		
DEC-OVMS	813	*y*	**Y**	**Y**		
Tandem	813	*y*	**Y**	**Y**		

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
# Only the ISO codeset on HP-UX is supported. The HP-UX proprietary greek8 codeset has no registered CCSID and is not supported.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 87. Conversion support: TURKISH

Target ▼ / Source →	OS/2, NT	AIX, HP-UX, Solaris, DEC-OVMS, Tandem	MVS, OS/400	Windows client	Apple client
CCSID	857	920	1026	1254	1281
MVS 1026	**Y†**	**Y†**	*y*	**Y†**	**Y†**
OS/400 1026	**Y**	**Y**	*y*	**Y**	**Y§**
OS/2 857	*y*	**Y††**	**Y**	**Y††**	**Y††**
AIX (ISO) 920	**Y**	*y*	**Y**	**Y††**	**Y††**
HP-UX (ISO) 920#	**Y**	*y*	**Y**	**Y††**	**Y††**
GIS					
NT 857	*y*	**Y**	**Y**	**Y††**	**Y††**
Solaris 920	**Y**	*y*	**Y**	**Y††**	**Y††**
SunOS					
SINIX, DC/OSx 920	**Y**	*y*	**Y**		
DEC-OVMS 920	**Y**	*y*	**Y**		
Tandem 920	**Y**	*y*	**Y**		

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
# Only the ISO codeset on HP-UX is supported. The HP-UX proprietary turkish8 codeset has no registered CCSID and is not supported.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 88. Conversion support: HEBREW

Target ↓	**Source** →	**MVS, OS/400**	**AIX**	**OS/2, NT**	**AIX, HP-UX, Solaris, DEC-OVMS, Tandem**	**Windows client**
	CCSID	424	856	862	916	1255
MVS	424	*y*	**Y†**	**Y†**	**Y†**	**Y†**
OS/400	424	*y*	**Y#**	**Y**	**Y**	**Y**
OS/2	862	**Y**	**Y††**	*y*	**Y††**	**Y††**
AIX (pc-A)	856	**Y+**	*y*	**Y+**	**Y+**	**Y††**
AIX (ISO)	916	**Y+**	**Y+**	**Y+**	*y*	**Y††**
HP-UX (ISO)	916&allt.	**Y**	**Y**	**Y**	*y*	**Y††**
GIS						
NT	862	**Y**	**Y**	*y*	**Y**	
Solaris	916	**Y**	**Y**	**Y**	*y*	**Y††**
SunOS						
SINIX, DC/OSx	916	**Y**	**Y**	**Y**	*y*	
DEC-OVMS	916	**Y**	**Y**	**Y**	*y*	
Tandem	916	**Y**	**Y**	**Y**	*y*	

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
# Only to/from CCSID 4952 (a variant of 856).
+ Only on AIX V4.1 and later.
§ Only the ISO codeset on HP-UX is supported. The HP-UX proprietary hebrew8 codeset has no registered CCSID and is not supported.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 89. Conversion support: ARABIC

Target ▼ / Source →	Source →	MVS, OS/400	OS/2, NT	AIX	AIX, HP-UX, Solaris, DEC-OVMS, Tandem	Windows client
	CCSID	420	864	1046	1089	1256
MVS	420	*y*	**Y†**	**Y†**	**Y†**	**Y†**
OS/400	420	*y*	**Y**	**Y**	**Y§§**	**Y**
OS/2	864	**Y**	*y*	**Y††**	**Y††**	**Y††**
AIX (pc-A)	1046	**Y#**	**Y#**	*y*	**Y#**	**Y††**
AIX (ISO)	1089	**Y#**	**Y#**	**Y#**	*y*	**Y††**
HP-UX (ISO)	1089§	**Y**	**Y**	**Y**	*y*	**Y††**
GIS						
NT	864	**Y**	*y*	**Y**	**Y**	**Y††**
Solaris	1089	**Y**	**Y**	**Y**	*y*	**Y††**
SunOS						
SINIX, DC/OSx	1089	**Y**	**Y**	**Y**	*y*	
DEC-OVMS	1089	**Y**	**Y**	**Y**	*y*	
Tandem	1089	**Y**	**Y**	**Y**	*y*	

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
# Only on AIX V4.1 and later.
§ Only the ISO codeset on HP-UX is supported. The HP-UX proprietary arabic8 codeset has no registered CCSID and is not supported.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§§ OS/400 V3R2, V3R7, V4R1 or later.

Table 90. Conversion support: FARSI

Target ↓	**Source** →	**MVS, OS/400**	**OS/2**
	CCSID	1097	1098
MVS	1097	*y*	**Y†**
OS/400	1097	*y*	**Y**
OS/2	1098	**Y††**	*y*
AIX	1098*	**Y††**	*y*
HP-UX	1098*	**Y††**	*y*
GIS (ISO)			
NT	1098*	**Y††**	*y*
Solaris	1098*	**Y††**	*y*
SunOS			
SINIX, DC/OSx			
DEC-OVMS			
Tandem			

Note:

***** The native CCSID for these platforms has not been standardized and may change.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 91. Conversion support: URDU

Target ▼ / Source →		OS/2, NT	MVS, OS/400	AIX, HP-UX, Solaris,
	CCSID	868	918	1006
MVS	918	**Y†**	*y*	
OS/400	918	**Y§§**	*y*	
OS/2	868	*y*	**Y††**	**Y††**
AIX	1006	**Y††**	**Y††**	*y*
HP-UX	1006	**Y††**	**Y††**	*y*
GIS (ISO)				
NT	868	*y*	**Y††**	**Y††**
Solaris	1006	**Y††**	**Y††**	*y*
SunOS				
SINIX, DC/OSx				
DEC-OVMS				
Tandem				

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.
§ OS/400 V3R2, V3R7, V4R1 or later.

Table 92. Conversion support: THAI

Target ↓	**Source** →	**MVS, OS/400**	**OS/2**
	CCSID	838	874
MVS	838	*y*	**Y†**
OS/400	838	*y*	**Y**
OS/2	874	**Y††**	*y*
AIX	874*	**Y††**	*y*
HP-UX	874*	**Y††**	*y*
GIS (ISO)			
NT	874*	**Y††**	*y*
Solaris	874*	**Y††**	*y*
SunOS			
SINIX, DC/OSx			
DEC-OVMS			
Tandem			

Note:

***** The native CCSID for these platforms has not been standardized and may change.
† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 93. Conversion support: JAPANESE LATIN SBCS

Target ↓ / Source →		OS/2, AIX, NT	OS/2	MVS, OS/400	AIX, Solaris
	CCSID	932	942	1027	5050 33722
MVS	1027			*y*	
OS/400	1027		**Y**	*y*	
OS/2	932	*y*	**Y**	**Y††**	**Y††**
OS/2	942	**Y**	*y*	**Y††**	**Y††**
AIX (pc-A)	932	*y*	**Y††**		**Y**
AIX (euc)	5050 33722*	**Y**	**Y††**		*y*
HP-UX					
GIS					
NT	932	*y*	**Y**	**Y**	**Y††**
Solaris	5050	**Y††**	**Y††**		*y*
SunOS					
SINIX, DC/OSx					
DEC-OVMS					
Tandem					

Note:

***** 5050 and 33722 are CCSIDs related to base code page 954 = eucJP on AIX. On AIX V3.2.5 MQSeries codes this code page as CCSID 5050 for compatibility with OS/400. On AIX V4.1 the CCSID reported by the operating system is 33722.

†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 94. Conversion support: JAPANESE KATAKANA SBCS

Target ↓	**Source** →	**MVS, OS/400**	**OS/2, HP-UX**	**AIX, NT**	**AIX, Solaris**
	CCSID	290	897	932	5050 33722
MVS	290	*y*	**Y**		
OS/400	290	*y*	**Y**		
OS/2	897	**Y**	*y*	**Y††**	**Y††**
AIX (pc-A)	932			*y*	**Y**
AIX (euc)	5050 33722*			**Y**	*y*
HP-UX (kana8)	897	**Y**	*y*		
GIS					
NT	932	**Y**	**Y**	*y*	**Y††**
Solaris	5050			**Y††**	*y*
SunOS					
SINIX, DC/OSx					
DEC-OVMS					
Tandem					

Note: In addition to the above conversions, MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later supports conversion from CCSID 897 to CCSIDs 37, 273, 277, 278, 280, 284, 285, 290, 297, 437, 500 819, 850, 1027 and 1252.

Note:

***** 5050 and 33722 are CCSIDs related to base code page 954 = eucJP on AIX. On AIX V3.2.5 MQSeries codes this code page as CCSID 5050 for compatibility with OS/400. On AIX V4.1 the CCSID reported by the operating system is 33722.

†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 95 (Page 1 of 2). Conversion support: JAPANESE KANJI / LATIN MIXED

Target ▼	**Source** →	**OS/2, AIX, HP-UX, DEC-OVMS, Tandem, NT**	**OS/2**	**NT**	**HP-UX, DEC-OVMS, Tandem**	**MVS, OS/400**	**AIX, Solaris**
	CCSID	932	942	943	954	5035	5050 33722
MVS	5035#	**Y†**	**Y†**			*y*	
OS/400	5035#	**Y**	**Y**	**Y§§**		*y*	**Y**
OS/2	932	*y*	**Y**	**Y††**	**Y††**	**Y**	**Y††**
OS/2	942	**Y**	*y*	**Y††**	**Y††**	**Y**	**Y††**
AIX (pc-A)	932	*y*	**Y††**	**Y††**	**Y**	**Y**	**Y**
AIX (ISO)	5050 33722*	**Y**	**Y††**	**Y††**	*y*	**Y**	*y*
HP-UX (euc)	954	**Y**			*y*	**Y****	*y*
HP-UX (-15§)	932	*y*			**Y**	**Y****	**Y**
GIS							
NT	932##	*y*	**Y**	**Y††**	**Y††**	**Y**	**Y††**
NT	943##	**Y††**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
Solaris	5050	**Y††**	**Y††**	**Y††**	**Y††**	**Y††**	*y*
SunOS							
SINIX, DC/OSx							
DEC-OVMS	932	*y*			**Y**	**Y**	**Y**
DEC-OVMS	954	**Y**			*y*	**Y**	*y*
Tandem	932	*y*			**Y**	**Y**	**Y**

Table 95 (Page 2 of 2). Conversion support: JAPANESE KANJI / LATIN MIXED

Target ↓	Source→	OS/2, AIX, HP-UX, DEC-OVMS, Tandem, NT	OS/2	NT	HP-UX, DEC-OVMS, Tandem	MVS, OS/400	AIX, Solaris
Tandem	954	**Y**			*y*	**Y**	*y*

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.

***** 5050 and 33722 are CCSIDs related to base code page 954 = eucJP on AIX. On AIX V3.2.5 MQSeries codes this code page as CCSID 5050 for compatibility with OS/400. On AIX V4.1 the CCSID reported by the operating system is 33722.

5035 is a CCSID related to code page 939.

§ Defined by HP-UX as japan15 and SJIS. Note that about 74 DBCS characters have different representations in japan15 and 932 so may not be converted correctly if the conversion is performed on a non-HP-UX system.

****** Supported on HP-UX V10 or later.

†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

§§ OS/400 V3R2, V3R6, V3R7, V4R1 or later.

NT uses the code page number 932, but this is best represented by the CCSID of 943. However not all platforms of MQSeries support this CCSID.

On versions of MQSeries for NT from version 5, CCSID 932 is used to represent code page 932, but a change to file `../conv/table/ccsid.tbl` can be made which changes the CCSID used to 943.

Table 96 (Page 1 of 2). Conversion support: JAPANESE KANJI / KATAKANA MIXED

Target ▼	**Source→**	**OS/2, AIX, HP-UX, DEC-OVMS, Tandem, NT**	**OS/2**	**NT**	**HP-UX, DEC-OVMS, Tandem**	**MVS, OS/400**	**AIX, Solaris**
	CCSID	932	942	943	954	5026	5050 33722
MVS	5026#	**Y†**	**Y†**			*y*	
OS/400	5026#	**Y**	**Y**	**Y§§**		*y*	**Y**
OS/2	932	*y*	**Y**	**Y††**	**Y††**	**Y**	**Y††**
OS/2	942	**Y**	*y*	**Y††**	**Y††**	**Y**	**Y††**
AIX (pc-A)	932	*y*	**Y††**	**Y††**	**Y**	**Y**	**Y**
AIX (euc)	5050 33722*	**Y**	**Y††**	**Y††**	*y*	**Y**	*y*
HP-UX (euc)	954	**Y**			*y*	**Y****	*y*
HP-UX (-15§)	932	*y*			**Y**	**Y**	**Y**
GIS							
NT	932##	*y*	**Y**	**Y††**	**Y††**	**Y**	**Y††**
NT	943##	**Y††**	**Y††**	*y*	**Y††**	**Y††**	**Y††**
Solaris	5050	**Y**	**Y††**	**Y††**	*y*	**Y**	*y*
SunOS							
SINIX, DC/OSx							
DEC-OVMS (sjis)	932	*y*			**Y**	**Y**	**Y**
DEC-OVMS (euc)	954	**Y**			*y*	**Y**	*y*
Tandem (sjis)	932	*y*		**Y**	**Y**	**Y**	

Table 96 (Page 2 of 2). Conversion support: JAPANESE KANJI / KATAKANA MIXED

Target ▼	Source→	OS/2, AIX, HP-UX, DEC-OVMS, Tandem, NT	OS/2	NT	HP-UX, DEC-OVMS, Tandem	MVS, OS/400	AIX, Solaris
Tandem (euc)	954	**Y**			*y*	**Y**	*y*

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.

***** 5050 and 33722 are CCSIDs related to base code page 954 = eucJP on AIX. On AIX V3.2.5 MQSeries codes this code page as CCSID 5050 for compatibility with OS/400. On AIX V4.1 the CCSID reported by the operating system is 33722.

5026 is a CCSID related to code page 930. CCSID 5026 is the CCSID reported to the user on OS/400 when the Japanese Katakana (DBCS) feature is selected.

§ Defined by HP-UX as japan15 and SJIS. Note that about 74 DBCS characters have different representations in japan15 and 932 so may not be converted correctly if the conversion is performed on a non-HP-UX system.

****** Supported on HP-UX V10 or later.

†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

§§ OS/400 V3R2, V3R6, V3R7, V4R1 or later.

NT uses the code page number 932, but this is best represented by the CCSID of 943. However not all platforms of MQSeries support this CCSID.

On versions of MQSeries for NT from version 5, CCSID 932 is used to represent code page 932, but a change to file `../conv/table/ccsid.tbl` can be made which changes the CCSID used to 943.

Table 97. Conversion support: KOREAN

Target ↓ / Source →		MVS, OS/400	OS/2, NT	AIX, HP-UX, DEC-OVMS, Tandem, Solaris
	CCSID	933	949	970
MVS	933	*y*	**Y†**	
OS/400	933	*y*	**Y**	**Y**
OS/2	949	**Y**	*y*	**Y††**
AIX (euc)	970	**Y**	**Y††**	*y*
HP-UX (-15)	949§	**Y**	*y*	
HP-UX (euc)	970§	**Y**		*y*
GIS				
NT	949	**Y**	*y*	**Y††**
Solaris	970	**Y††**	**Y††**	*y*
SunOS				
SINIX, DC/OSx				
DEC-OVMS	970	**Y**	**Y**	*y*
Tandem	970	**Y**	**Y**	*y*

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
§ On HP-UX9 949 is used, but on HP-UX10 970 is used.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 98. Conversion support: SIMPLIFIED CHINESE

Target ↓	Source→	MVS, OS/400	OS/2, HP-UX, NT	AIX, DEC-OVMS, Tandem Solaris	NT	MVS
	CCSID	935	1381	1383	1386	1388
MVS	935	*y*	**Y†**			
MVS	1388					
OS/400	935	*y*	**Y**	**Y+**	**Y§§**	**Y§§**
OS/2	1381	**Y**	*y*	**Y††**	**Y††**	**Y††**
AIX (euc)	1383*	**Y***	**Y***	*y*	**Y††**	**Y††**
HP-UX (-15)	1381§	**Y****	*y*			
GIS						
NT	1381##	**Y**	*y*	**Y††**	**Y††**	**Y††**
NT	1386##	**Y**	**Y††**	**Y††**	*y*	**Y††**
Solaris	1383	**Y††**	**Y††**	*y*	**Y††**	**Y††**
SunOS						
SINIX, DC/OSx						
DEC-OVMS	1383	**Y**	**Y**	*y*		
Tandem	1383	**Y**	**Y**	*y*		

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
+ Supported on OS/400 V3R7 or later.
***** Supported on country AIX version only.
§ Is called prc15 and hp15CN on HP-UX.
****** Supported on HP-UX V10 or later.
§§ OS/400 V3R2, V3R6, V3R7, V4R1 or later.
NT uses the code page number 936, but this is best represented by the CCSID of 1386. However not all platforms of MQSeries support this CCSID.

On versions of MQSeries for NT prior to version 5, CCSID 1381 is used to represent code page 936.

On versions of MQSeries for NT from version 5, CCSID 1381 is used to represent code page 936, but a change to file `../conv/table/ccsid.tbl` can be made which changes the CCSID used to 1386.

†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 99 (Page 1 of 2). Conversion support: TRADITIONAL CHINESE

Target ▼	**Source** →	**MVS, OS/400**	**OS/2, HP-UX**	**OS/2**	**OS/2, AIX, HP-UX, NT, DEC-OVMS, Tandem**	**AIX, HP-UX, DEC-OVMS, Tandem, Solaris**
	CCSID	937	938	948	950	964
MVS	937	*y*	**Y†**	**Y†**	**Y†**	
OS/400	937	*y*	**Y**	**Y**	**Y**	**Y**
OS/2 (PS/55)	938	**Y**	*y*	**Y††**	**Y††**	**Y††**
OS/2 (PS/55)	948	**Y**	**Y††**	*y*	**Y††**	**Y††**
OS/2 (big5)	950	**Y**	**Y††**	**Y††**	*y*	**Y††**
AIX (euc)	964	**Y**	**Y**	**Y††**	**Y**	*y*
AIX (big5)	950	**Y**	**Y**	**Y††**	*y*	**Y**
HP-UX (-15§)	938	**Y**	*y*		**Y**	**Y****
HP-UX (big5)	950	**Y****	**Y**		*y*	**Y****
HP-UX (eucTW)	964	**Y****	**Y****		**Y****	*y*
GIS						
NT	950	**Y**	**Y**	**Y**	*y*	**Y††**
Solaris	964	**Y††**		**Y††**	**Y††**	*y*
SunOS						
SINIX, DC/OSx						
DEC-OVMS (euc)	964	**Y**	**Y**	**Y**	**Y**	*y*
DEC-OVMS (big5)	950	**Y**	**Y**	**Y**	*y*	**Y**
Tandem (euc)	964	**Y**	**Y**	**Y**	**Y**	*y*

Table 99 (Page 2 of 2). Conversion support: TRADITIONAL CHINESE

Target ↓	**Source** →	**MVS, OS/400**	**OS/2, HP-UX**	**OS/2**	**OS/2, AIX, HP-UX, NT, DEC-OVMS, Tandem**	**AIX, HP-UX, DEC-OVMS, Tandem, Solaris**
Tandem (big5)	950	**Y**	**Y**	**Y**	*y*	**Y**

Note:

† Supported on MQSeries for MVS/ESA version 1.1.4 or later.
§ Is called roc15 and eucTW on HP-UX.
****** Supported on HP-UX V10 or later.
†† Supported on MQSeries for AIX, OS2 Warp, HP-UX, Sun Solaris, or Windows NT Version 5 or later.

Table 100 (Page 1 of 6). MVS/ESA V1.1.4 or later single byte CCSID conversion support.	
CCSID	**Converts to and from CCSIDS**
37	256, 273, 275, 277, 278, 280, 284, 285, 290, 297, 367, 420, 423, 424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903-905, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1097, 1100, 1114, 1252, 1275
256	37, 273, 277, 278, 280, 284, 285, 290, 297, 367, 420, 423, 424, 437, 500, 819, 833, 836, 838, 850, 852, 857, 860-866, 869-871, 875, 880, 905, 1025-1027, 1251, 1252, 1275
259	437, 850-852, 855-857, 860-865, 869, 874, 899, 915, 1098, 1251
273	37, 256, 277, 278, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855-857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
274	500, 1047
275	37, 500, 1047
277	37, 256, 273, 278, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
278	37, 256, 273, 277, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
280	37, 256, 273, 277, 278, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
281	1047
282	500, 1047
284	37, 256, 273, 277, 278, 280, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
285	37, 256, 273, 277, 278, 280, 284, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
290	37, 256, 273, 277, 278, 280, 284, 285, 297, 367, 437, 500, 819, 833, 836, 850, 852, 855, 857, 860-865, 870, 871, 895-897, 1009, 1025-1027, 1040-1043, 1088
297	37, 256, 273, 277, 278, 280, 284, 285, 290, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1100, 1252, 1275
367	37, 256, 273, 277, 278, 280, 284, 290, 297, 500, 833, 836, 871, 875, 1009, 1026, 1027, 1041, 1088, 1115
420	37, 256, 424, 437, 500, 819, 850, 852, 857, 860-865, 1008, 1046, 1089, 1098, 1256
423	37, 256, 273, 277, 278, 280, 284, 285, 297, 437, 500, 813, 819, 838, 850-852, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1041-1043, 1253, 1280
424	37, 256, 420, 437, 500, 803, 819, 836, 850, 852, 856, 857, 860-865, 916, 1255

Table 100 (Page 2 of 6). MVS/ESA V1.1.4 or later single byte CCSID conversion support.

CCSID	Converts to and from CCSIDS
437	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-863, 865, 866, 869-871, 874, 875, 880, 897, 903, 905, 912, 915, 916, 920, 1025-1027, 1040-1043, 1051, 1097, 1098, 1252, 1275, 4946, 28709
500	37, 256, 273-275, 277, 278, 280, 282, 284, 285, 290, 297, 367, 420, 423, 424, 437, 813, 819, 833, 836, 838, 850-852, 855-857, 860-866, 869-871, 874, 875, 880, 891, 895, 897, 903-905, 912, 915, 916, 920, 1004, 1009-1021, 1023, 1025-1027, 1040-1043, 1046, 1047, 1051, 1088, 1089, 1097, 1100-1107, 1114, 1115, 1250-1256, 1275
803	424, 856, 862, 916
813	37, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 819, 838, 850, 852, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1253, 1280
819	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 833, 836, 838, 850, 852, 857, 860, 861, 863, 865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1047, 1051, 1097, 1098, 1114, 1252, 1275
833	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 367, 437, 500, 819, 836, 850, 852, 855, 857, 860-865, 870, 871, 891, 1009, 1025-1027, 1040-1043, 1088
836	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 367, 424, 437, 500, 819, 833, 850, 852, 855, 857, 870, 871, 875, 903, 1009, 1025-1027, 1040-1043, 1088, 1115
838	37, 256, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 850, 852, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043
850	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 819, 833, 836, 838, 852, 855-857, 860-866, 869-871, 874, 875, 880, 897, 903, 905, 912, 915, 916, 920, 1025-1027, 1040-1043, 1047, 1051, 1088, 1097, 1098, 1100, 1114, 1252, 1275, 4953
851	259, 423, 500, 875
852	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 819, 833, 836, 838, 850, 855, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1088, 1097, 1250, 1282, 28709
855	37, 259, 273, 277, 278, 280, 284, 285, 290, 297, 437, 500, 833, 836, 850, 852, 857, 866, 870, 871, 880, 912, 915, 1025-1027, 1040-1043, 1088, 1251, 1283
856	259, 273, 424, 500, 803, 850, 862, 916, 1255
857	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1040-1043, 1088, 1097, 1254, 1281, 28709
860	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 819, 833, 838, 850, 852, 857, 861, 863, 865, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1041-1043, 1097, 28709
861	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 819, 833, 838, 850, 852, 857, 860, 863, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1041-1043, 1097, 28709

Table 100 (Page 3 of 6). MVS/ESA V1.1.4 or later single byte CCSID conversion support.	
CCSID	**Converts to and from CCSIDS**
862	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 803, 833, 838, 850, 856, 870, 871, 875, 880, 905, 916, 1025-1027, 1097, 1255, 28709
863	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 813, 819, 833, 838, 850, 852, 857, 860, 861, 865, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 920, 1025-1027, 1041-1043, 1051, 1097, 1252, 1275, 28709
864	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 500, 833, 838, 850, 870, 871, 875, 880, 905, 918, 1008, 1025-1027, 1046, 1089, 1097, 1256, 28709
865	37, 256, 259, 273, 277, 278, 280, 284, 285, 290, 297, 420, 423, 424, 437, 500, 819, 833, 838, 850, 860, 863, 870, 871, 875, 880, 905, 1025-1027, 1097, 28709
866	256, 437, 500, 850, 855, 870, 880, 915, 1025, 1251, 1283
868	918
869	37, 256, 259, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860, 861, 863, 870, 871, 874, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043, 1253, 1254, 1280
870	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-866, 869, 871, 874, 875, 880, 897, 903, 912, 915, 916, 920, 1009, 1025-1027, 1040-1043, 1088, 1250, 1282
871	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869, 870, 874, 875, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1040-1043, 1047, 1051, 1088, 1252, 1275
874	37, 259, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860, 861, 863, 869-871, 875, 880, 897, 903, 912, 916, 920, 1025-1027, 1041-1043
875	37, 256, 273, 277, 278, 280, 284, 285, 297, 367, 423, 437, 500, 813, 819, 836, 838, 850-852, 857, 860-865, 869-871, 874, 880, 897, 903, 912, 916, 920, 1009, 1025-1027, 1041-1043, 1047, 1088, 1253, 1280
880	37, 256, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 855, 857, 860-866, 869-871, 874, 875, 897, 903, 912, 915, 916, 920, 1009, 1025-1027, 1041-1043, 1251, 1283
891	500, 833, 1088
895	290, 500, 1027, 1041
896	290, 1027, 1041
897	37, 273, 277, 278, 280, 284, 285, 290, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860, 861, 863, 869-871, 874, 875, 880, 903, 912, 916, 920, 1025-1027, 1041-1043
899	259
903	37, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 836, 838, 850, 852, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 912, 916, 920, 1025-1027, 1041-1043, 1115

Table 100 (Page 4 of 6). MVS/ESA V1.1.4 or later single byte CCSID conversion support.

CCSID	Converts to and from CCSIDS
904	37, 500, 1114
905	37, 256, 437, 500, 850, 852, 857, 860-865, 920, 1026, 1254, 1281
912	37, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 855, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 916, 920, 1025-1027, 1041-1043, 1250, 1282
915	259, 437, 500, 850, 855, 866, 870, 880, 1025, 1251, 1283
916	37, 273, 277, 278, 280, 284, 285, 297, 423, 424, 437, 500, 803, 813, 819, 838, 850, 852, 856, 857, 860-863, 869-871, 874, 875, 880, 897, 903, 912, 920, 1025-1027, 1041-1043, 1255
918	864, 868
920	37, 273, 277, 278, 280, 284, 285, 297, 423, 437, 500, 813, 819, 838, 850, 852, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 1025, 1026, 1254, 1281
1004	500
1008	420, 864
1009	37, 273, 277, 278, 280, 284, 290, 297, 367, 423, 500, 833, 836, 870, 871, 875, 880, 1025, 1026
1010	500
1011	500
1012	500
1013	500
1014	500
1015	500
1016	500
1017	500
1018	500
1019	500
1020	500
1021	500
1023	500
1025	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-866, 869-871, 874, 875, 880, 897, 903, 912, 915, 916, 920, 1009, 1026, 1027, 1040-1043, 1051, 1088, 1251, 1283

Table 100 (Page 5 of 6). MVS/ESA V1.1.4 or later single byte CCSID conversion support.	
CCSID	**Converts to and from CCSIDS**
1026	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 897, 903, 905, 912, 916, 920, 1009, 1025, 1027, 1040-1043, 1047, 1088, 1254, 1281
1027	37, 256, 273, 277, 278, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860-865, 869-871, 874, 875, 880, 895-897, 903, 912, 916, 1025, 1026, 1040-1043, 1047, 1088
1040	37, 273, 277, 278, 280, 284, 285, 290, 297, 437, 500, 833, 836, 850, 852, 855, 857, 870, 871, 1025-1027, 1041-1043, 1088
1041	37, 273, 277, 278, 280, 284, 285, 290, 297, 367, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860, 861, 863, 869-871, 874, 875, 880, 895-897, 903, 912, 916, 1025-1027, 1040, 1042, 1043, 1088
1042	37, 273, 277, 278, 280, 284, 285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 912, 916, 1025-1027, 1040, 1041, 1043, 1088
1043	37, 273, 277, 278, 280, 284, 285, 290, 297, 423, 437, 500, 813, 819, 833, 836, 838, 850, 852, 855, 857, 860, 861, 863, 869-871, 874, 875, 880, 897, 903, 912, 916, 1025-1027, 1040-1042, 1088, 1114
1046	420, 500, 864, 1089, 1256
1047	37, 273-275, 277, 278, 280-282, 284, 285, 297, 500, 819, 850, 871, 875, 1026, 1027
1051	37, 273, 277, 278, 280, 284, 285, 297, 437, 500, 819, 850, 863, 871, 1025, 1097, 1252, 1275
1088	37, 273, 277, 278, 280, 284, 285, 290, 297, 367, 500, 833, 836, 850, 852, 855, 857, 870, 871, 875, 891, 1025-1027, 1040-1043
1089	420, 500, 864, 1046, 1256
1097	37, 437, 500, 819, 850, 852, 857, 860-865, 1051, 1098
1098	259, 420, 437, 819, 850, 1097
1100	37, 273, 277, 278, 280, 284, 285, 297, 500, 850
1101	500
1102	500
1103	500
1104	500
1105	500
1106	500
1107	500
1114	37, 500, 819, 850, 904, 1043
1115	367, 500, 836, 903

Table 100 (Page 6 of 6). MVS/ESA V1.1.4 or later single byte CCSID conversion support.	
CCSID	**Converts to and from CCSIDS**
1250	500, 852, 870, 912, 1282
1251	256, 259, 500, 855, 866, 880, 915, 1025, 1283
1252	37, 256, 273, 277, 278, 280, 284, 285, 297, 437, 500, 819, 850, 863, 871, 1051, 1275
1253	423, 500, 813, 869, 875, 1280
1254	500, 857, 869, 905, 920, 1026, 1281
1255	424, 500, 856, 862, 916
1256	420, 500, 864, 1046, 1089
1275	37, 256, 273, 277, 278, 280, 284, 285, 297, 437, 500, 819, 850, 863, 871, 1051, 1252
1280	423, 813, 869, 875, 1253
1281	857, 905, 920, 1026, 1254
1282	852, 870, 912, 1250
1283	855, 866, 880, 915, 1025, 1251
4946	437
4953	850
28709	437, 852, 857, 860-865

OS/2 conversion support

MQSeries for OS/2 Warp V5 or later supports conversion between any of the CCSIDS listed below:

037	256	259	273	274	277
278	280	282	284	285	287
290	293	297	300	301	361
363	367	382	383	385	386
387	388	389	391	392	393
394	395	420	423	424	437
500	813	819	829	833	834
835	836	837	838	850	851
852	855	856	857	860	861
862	863	864	865	866	868
869	870	871	874	875	880
891	895	896	897	903	904
905	907	909	910	912	913
914	915	916	918	919	920
921	922	927	930	932	933
935	938 (1)	937	939	941	942
943	946	947	948	949	950
951	952	954 (2)	955	960	961
963	964	970	971	1004	1006
1008	1009	1010	1011	1012	1013
1014	1015	1016	1017	1018	1019
1025	1026	1027	1028	1038	1040
1041	1042	1043	1046	1047	1050
1051	1088	1089	1092	1097	1098
1112	1114	1115	1116	1117	1118
1119	1122	1123	1124	1200	1208
1250	1251	1252	1253	1254	1255
1256	1257	1275	1276	1277	1350
1380	1381	1382	1383	1386	1388
4948	4951	4952	4960	5026	5035
5037	5039	5048	5049	5050 (2)	5067
5142	5478	8612	9030	9056	9066
9145	13488	28709	33722		

Notes:

1. – 938 uses 948 for conversion.
2. – 954 and 5050 use 33722 for conversion.

OS/400 conversion support

A full list of CCSIDs, and conversions supported by OS/400, can be found in the appropriate AS/400 publication relating to your operating system.

Unicode conversion support

Some platforms support the conversion of user data to or from Unicode encoding. The two forms of unicode encoding supported are UCS-2 (CCSIDs 1200 and 13488) and UTF-8 (CCSID 1208).

Note: MQSeries does not support queue manager Unicode CCSIDs so message header data cannot be encoded in UNICODE.

MQSeries OS/2 support for Unicode

On MQSeries for OS/2 Warp V5 or later, conversion on OS/2 to and from the Unicode CCSIDs is supported for all supported CCSIDs. See “OS/2 conversion support” on page 567

MQSeries AIX support for Unicode

On MQSeries for AIX Version 5 or later, conversion on AIX to and from the Unicode CCSIDs is supported for the following CCSIDs:

037	273	278	280	284	285
297	423	437	500	813	819
850	852	856	857	860	861
865	869	875	880	912	915
916	920	932	933	935	937
938	939	939	942	943	948
949	950	954	964	970	1026
1046	1089	1131	1200	1208	1250
1251	1253	1254	1280	1281	1282
1283	1284	1285	1381	1383	1386
1388	5026	5035	5050	13488	33722

MQSeries HP-UX support for Unicode

On MQSeries for HP-UX Version 5 or later, conversion on HP to, and from, the Unicode CCSIDs is supported for the following CCSIDs:

813	819	874	912	915	916
920	932	938	950	954	964
970	1051	1089	1200	1381	5050
13488	33722				

Note: HP-UX does not support conversion into or from UTF-8.

MQSeries NT and Solaris support for Unicode

On MQSeries for Windows NT Version 5 or later, and MQSeries for Solaris 5 or later, conversion to, and from, the Unicode CCSIDs is supported for the following CCSIDs:

037	277	278	280	284	285
290	297	300	301	420	424
437	500	813	819	833	835
836	837	838	850	852	855
856	857	860	861	862	863
864	865	866	868	869	870
871	874	875	880	891	897
903	904	912	915	916	918
920	921	922	927	928	930
931 (1)	932 (2)	933	935	937	938 (3)
939	941	942	943	947	948
949	950	951	954 (4)	964	970
1006	1025	1026	1027	1040	1041
1042	1043	1046	1047	1051	1088
1089	1097	1098	1112	1114	1115
1122	1123	1124	1200	1208	1250
1251	1252	1253	1254	1255	1256
1257	1275	1280	1281	1282	1283
1380	1381	1383	1386	1388	5050
13488	33722 (4)				

Notes:

1. – 931 uses 939 for conversion.
2. – 932 uses 942 for conversion.
3. – 938 uses 948 for conversion.
4. – 954 and 33722 use 5050 for conversion.

OS/400 support for Unicode

OS/400 supports a special variant of UNICODE with CCSID 61952 from Version 3.1 onwards. Version 3.7 and later versions also support UNICODE CCSID 13488.

Appendix G. 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. Subject to IBM’s valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service 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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact Laboratory Counsel, MP151, IBM United Kingdom Laboratories, Hursley Park, Winchester, Hampshire, England SO21 2JN. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

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 Avenue, Thornwood, New York 10594, U.S.A.

Programming interface information

This book is intended to help you to write application programs provided by MQSeries products.

This book documents General-use Programming Interface and Associated Guidance Information and Product-sensitive Programming Interface and Associated Guidance Information provided by MQSeries.

General-use programming interfaces allow the customer to write programs that obtain the services of these products.

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of these products. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive programming interfaces should be used only for these specialized purposes.

Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive programming interface

Product-sensitive Programming Interface and Associated Guidance Information...

End of Product-sensitive programming interface

Trademarks

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

AD/Cycle	AIX/6000	AS/400
BookManager	CICS	CICS for MVS/ESA
COBOL/370	Common User Access	DB2
IBM	Micro Focus	MQSeries
MVS/ESA	OS/2	OS/400
SAA	System/370	System/390
VSE/ESA		

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.

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

Java and HotJava are trademarks of Sun Microsystems, Inc.

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

Glossary of terms and abbreviations

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

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

A

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

active log. See *recovery log*.

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

add-in task. A function provided by MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT that coordinates the passing of data between a Lotus Notes application and an MQSeries application.

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

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

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

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

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

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

allied address space. See *ally*.

ally. An MVS address space that is connected to MQSeries for MVS/ESA.

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

APAR. Authorized program analysis report.

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

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

application queue. A queue used by an application.

archive log. See *recovery log*.

ASID. Address space identifier.

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

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

authorization checks. Security checks that are performed when a user tries to open an MQSeries object.

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

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

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

B

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

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

BMS. Basic mapping support.

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

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

The BSDS is required if the MQSeries for MVS/ESA subsystem has to be restarted.

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

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

BSDS. Bootstrap data set.

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

C

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

CCF. Channel control function.

CCSID. Coded character set identifier.

CDF. Channel definition file.

channel. See *message channel*.

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

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

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

checkpoint. (1) A time when significant information is written on the log. Contrast with *syncpoint*. (2) In MQSeries on UNIX systems, the point in time when a data record described in the log is the same as the data record in the queue. Checkpoints are generated automatically and are used during the system restart process.

CI. Control interval.

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

CL. Control Language.

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

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

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

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

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

command prefix (CPF). In MQSeries for MVS/ESA, a character string that identifies the queue manager to which MQSeries for MVS/ESA commands are directed,

and from which MQSeries for MVS/ESA operator messages are received.

command processor. The MQSeries component that processes commands.

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

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

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

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

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

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

context. Information about the origin of a message.

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

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

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

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

controlled shutdown. See *quiesced shutdown*.

CPF. Command prefix.

D

DAE. Dump analysis and elimination.

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

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

DCE. Distributed Computing Environment.

DCI. Data conversion interface.

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

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

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

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

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

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

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

DLQ. Dead-letter queue.

DQM. Distributed queue management.

dual logging. A method of recording MQSeries for MVS/ESA activity, where each change is recorded on two data sets, so that if a restart is necessary and one

data set is unreadable, the other can be used. Contrast with *single logging*.

dual mode. See *dual logging*.

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

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

E

environment. See *application environment*.

ESM. External security manager.

ESTAE. Extended specify task abnormal exit.

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

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

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

event log. See *application log*.

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

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

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

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

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

F

FFST. First Failure Support Technology.

FIFO. First-in-first-out.

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

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

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

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

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

FRR. Functional recovery routine.

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

G

GCPC. Generalized command preprocessor.

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

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

get. In message queuing, to use the MQGET call to remove a message from a queue. See also *browse*.

global trace. An MQSeries for MVS/ESA trace option where the trace data comes from the entire MQSeries for MVS/ESA subsystem.

GTF. Generalized Trace Facility.

H

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

I

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

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

.ini file. See *configuration file*.

initialization input data sets. Data sets used by MQSeries for MVS/ESA when it starts up.

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

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

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

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

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

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

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

IPCS. Interactive Problem Control System.

ISPF. Interactive System Productivity Facility.

L

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

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

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

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

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

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

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

log. In MQSeries, a file recording the work done by queue managers while they receive, transmit, and deliver messages.

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

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

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

M

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

mail-in database. A Lotus Notes database for sole use by the add-in task. It holds the request from a Lotus Notes application before the request is passed to the MQSeries application.

MCA. Message channel agent.

MCI. Message channel interface.

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

message. (1) In message queuing applications, a communication sent between programs. See also *persistent message* and *nonpersistent message*. (2) In system programming, information intended for the terminal operator or system administrator.

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

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

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

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

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

message queue. Synonym for *queue*.

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

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

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

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

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

MQI. Message queue interface.

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

MQSC. MQSeries commands.

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

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

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects. Contrast with *programmable command format (PCF)*.

N

namelist. An MQSeries for MVS/ESA object that contains a list of queue names.

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

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

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

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

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

NSI. Name service interface.

NTFS. New Technology File System.

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

O

OAM. Object authority manager.

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

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

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

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

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

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

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

P

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

PCF. Programmable command format.

PCF command. See *programmable command format*.

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

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

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

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

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

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

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

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

point of recovery. In MQSeries for MVS/ESA, the term used to describe a set of backup copies of MQSeries for MVS/ESA page sets and the corresponding log data sets required to recover these page sets. These backup copies provide a potential

restart point in the event of page set loss (for example, page set I/O error).

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

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

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

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

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

Contrast with *MQSC*.

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

PTF. Program temporary fix.

Q

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

queue manager. (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.

queue manager event. An event that indicates:

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

queuing. See *message queuing*.

quiesced shutdown. (1) In MQSeries, a shutdown of a queue manager that allows all connected applications to disconnect. Contrast with *immediate shutdown* and *preemptive shutdown*. (2) A type of shutdown of the CICS adapter where the adapter disconnects from MQSeries, but only after all the currently active tasks have been completed. Contrast with *forced shutdown*.

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

R

RBA. Relative byte address.

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

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

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

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

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

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

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

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

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

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

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

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

reply message. A type of message used for replies to request messages.

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

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

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

request message. A type of message used to request a reply from another program.

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

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

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

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

responder. In distributed queuing, a program that replies to network connection requests from another system.

resynch. In MQSeries, an option to direct a channel to start up and resolve any in-doubt status messages, but without restarting message transfer.

return codes. The collective name for completion codes and reason codes.

rollback. Synonym for *back out*.

RTM. Recovery termination manager.

rules table. A control file containing one or more rules that the dead-letter queue handler applies to messages on the DLQ.

S

SAF. System Authorization Facility.

SDWA. System diagnostic work area.

security enabling interface (SEI). The MQSeries interface to which customer- or vendor-written programs that check authorization, supply a user identifier, or perform authentication must conform. A part of the MQSeries Framework.

SEI. Security enabling interface.

sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

sequential delivery. In MQSeries, a method of transmitting messages with a sequence number so that the receiving channel can reestablish the message sequence when storing the messages. This is required where messages must be delivered only once, and in the correct order.

sequential number wrap value. In MQSeries, a method of ensuring that both ends of a communication link reset their current message sequence numbers at the same time. Transmitting messages with a sequence number ensures that the receiving channel can reestablish the message sequence when storing the messages.

server. (1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *client*.

server channel. In message queuing, a channel that responds to a requester channel, removes messages

from a transmission queue, and moves them over a communication link to the requester channel.

server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *client connection channel type*.

service interval. A time interval, against which the elapsed time between a put or a get and a subsequent get is compared by the queue manager in deciding whether the conditions for a service interval event have been met. The service interval for a queue is specified by a queue attribute.

service interval event. An event related to the service interval.

session ID. In MQSeries for MVS/ESA, the CICS-unique identifier that defines the communication link to be used by a message channel agent when moving messages from a transmission queue to a link.

shutdown. See *immediate shutdown*, *preemptive shutdown*, and *quiesced shutdown*.

signaling. In MQSeries for MVS/ESA and MQSeries for Windows 2.1, a feature that allows the operating system to notify a program when an expected message arrives on a queue.

single logging. A method of recording MQSeries for MVS/ESA activity where each change is recorded on one data set only. Contrast with *dual logging*.

single-phase backout. A method in which an action in progress must not be allowed to finish, and all changes that are part of that action must be undone.

single-phase commit. A method in which a program can commit updates to a queue without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with *two-phase commit*.

SIT. System initialization table.

stanza. A group of lines in a configuration file that assigns a value to a parameter modifying the behavior of a queue manager, client, or channel. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a configuration (.ini) file may contain a number of stanzas.

storage class. In MQSeries for MVS/ESA, a storage class defines the page set that is to hold the messages for a particular queue. The storage class is specified when the queue is defined.

store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

subsystem. In MVS, a group of modules that provides function that is dependent on MVS. For example, MQSeries for MVS/ESA is an MVS subsystem.

supervisor call (SVC). An MVS instruction that interrupts a running program and passes control to the supervisor so that it can perform the specific service indicated by the instruction.

SVC. Supervisor call.

switch profile. In MQSeries for MVS/ESA, a RACF profile used when MQSeries starts up or when a refresh security command is issued. Each switch profile that MQSeries detects turns off checking for the specified resource.

symptom string. Diagnostic information displayed in a structured format designed for searching the IBM software support database.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

syncpoint. An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

System Authorization Facility (SAF). An MVS facility through which MQSeries for MVS/ESA communicates with an external security manager such as RACF.

system.command.input queue. A local queue on which application programs can put MQSeries commands. The commands are retrieved from the queue by the command server, which validates them and passes them to the command processor to be run.

system control commands. Commands used to manipulate platform-specific entities such as buffer pools, storage classes, and page sets.

system diagnostic work area (SDWA). Data recorded in a SYS1.LOGREC entry, which describes a program or hardware error.

system initialization table (SIT). A table containing parameters used by CICS on start up.

SYS1.LOGREC. A service aid containing information about program and hardware errors.

T

TACL. Tandem Advanced Command Language.

target library high-level qualifier (thlqual). High-level qualifier for MVS/ESA target data set names.

task control block (TCB). An MVS control block used to communicate information about tasks within an address space that are connected to an MVS subsystem such as MQSeries for MVS/ESA or CICS.

task switching. The overlapping of I/O operations and processing between several tasks. In MQSeries for MVS/ESA, the task switcher optimizes performance by allowing some MQI calls to be executed under subtasks rather than under the main CICS TCB.

TCB. Task control block.

temporary dynamic queue. A dynamic queue that is deleted when it is closed. Temporary dynamic queues are not recovered if the queue manager fails, so they can contain nonpersistent messages only. Contrast with *permanent dynamic queue*.

termination notification. A pending event that is activated when a CICS subsystem successfully connects to MQSeries for MVS/ESA.

thlqual. Target library high-level qualifier.

thread. In MQSeries, the lowest level of parallel execution available on an operating system platform.

time-independent messaging. See *asynchronous messaging*.

TMI. Trigger monitor interface.

trace. In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF). See also *global trace* and *performance trace*.

tranid. See *transaction identifier*.

transaction identifier. In CICS, a name that is specified when the transaction is defined, and that is used to invoke the transaction.

transmission program. See *message channel agent*.

transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message. A message containing information about the program that a trigger monitor is to start.

trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

trigger monitor interface (TMI). The MQSeries interface to which customer- or vendor-written trigger monitor programs must conform. A part of the MQSeries Framework.

two-phase commit. A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with *single-phase commit*.

U

UIS. User identifier service.

undelivered-message queue. See *dead-letter queue*.

undo/redo record. A log record used in recovery. The redo part of the record describes a change to be made to an MQSeries object. The undo part describes how to back out the change if the work is not committed.

unit of recovery. A recoverable sequence of operations within a single resource manager. Contrast with *unit of work*.

unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Contrast with *unit of recovery*.

user identifier service (UIS). In MQSeries for OS/2 Warp, the facility that allows MQI applications to associate a user ID, other than the default user ID, with MQSeries messages.

utility. In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Sending your comments to IBM

MQSeries

Application Programming Reference

SC33-1673-04

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To 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., after your international access code use 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: 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 to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

MQSeries

Application Programming Reference

SC33-1673-04

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	
Telephone	Email

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.).

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
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name _____

Company or Organization _____

Address _____

_____

EMAIL _____

Telephone _____

4 Fasten here with adhesive tape

Cut along this line

Cut along this line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC33-1673-04

Spine information:

MQSeries Application Programming Reference