

MQSeries[®] for AS/400[®]



Application Programming Reference (ILE RPG)

V5.1

MQSeries[®] for AS/400[®]



Application Programming Reference (ILE RPG)

V5.1

Note!

Before using this information and the product it supports, be sure to read the general information under “Appendix G. Notices” on page 483.

First edition (March 2000)

This edition applies to MQSeries for AS/400 Version 5 Release 1 and to any subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright International Business Machines Corporation 1994, 2000. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables	ix
------------------	----

About this book	xi
----------------------------------	-----------

Who this book is for	xi
What you need to know to understand this book	xi
How to use this book	xii
Appearance of text in this book	xii
Terms used in this book	xii

Part 1. Data type descriptions 1

Chapter 1. Elementary data types. . . . 3

Conventions used in the descriptions of data types	3
Elementary data types	3
MQBYTE - Byte	3
MQBYTEn - String of n bytes	4
MQCHAR - character	4
MQCHARn - String of n characters	4
MQHCONN - Connection handle	4
MQHOBJ - Object handle	5
MQLONG - Long integer	5
Elementary data types	5

Chapter 2. Structure data types – programming considerations 7

Conventions used in the descriptions of data types	7
Language considerations	8
COPY files	8
Calls	9
Structures	10
Notational conventions	10
MQI procedures	10
Threading considerations	10
MQI call parameters	11
Named constants	11
Commitment control	11
Coding the bound calls	11
Coding the dynamic calls	12

Chapter 3. MQBO - Begin options . . . 15

Overview	15
Fields	15
Initial values and RPG declaration	16
RPG declaration	16

Chapter 4. MQCIH - CICS bridge header 17

Overview	18
Fields	19
Initial values and RPG declaration	28
RPG declaration	29

Chapter 5. MQCNO - Connect options 31

Overview	31
--------------------	----

Fields	31
Initial values and RPG declaration	35
RPG declaration	35

Chapter 6. MQDH - Distribution header 37

Overview	37
Fields	38
Initial values and RPG declaration	41
RPG declaration	41

Chapter 7. MQDLH - Dead-letter header 43

Overview	43
Fields	45
Initial values and RPG declaration	49
RPG declaration	49

Chapter 8. MQGMO - Get-message options 51

Overview	51
Fields	51
Initial values and RPG declaration	74
RPG declaration	74

Chapter 9. MQIIH - IMS bridge header 77

Overview	77
Fields	78
Initial values and RPG declaration	81
RPG declaration	82

Chapter 10. MQMD - Message descriptor 83

Overview	84
Fields	85
Initial values and RPG declaration	129
RPG declaration	130

Chapter 11. MQMDE - Message descriptor extension 131

Overview	131
Fields	133
Initial values and RPG declaration	135
RPG declaration	136

Chapter 12. MQOD - Object descriptor 137

Overview	137
Fields	138
Initial values and RPG declaration	145
RPG declaration	146

Chapter 13. MQOR - Object record 147

Overview	147
Fields	147
Initial values and RPG declaration	148

RPG declaration	148
Chapter 14. MQPMO - Put message options	149
Overview.	149
Fields	150
Initial values and RPG declaration	163
RPG declaration	164
Chapter 15. MQPMR - Put-message record	165
Overview.	165
Fields	165
Initial values and RPG declaration	167
RPG declaration	167
Chapter 16. MQRMH - Message reference header	169
Overview.	169
Fields	170
Initial values and RPG declaration	175
RPG declaration	176
Chapter 17. MQRR - Response record	177
Overview.	177
Fields	177
Initial values and RPG declaration	177
RPG declaration	178
Chapter 18. MQTM - Trigger message	179
Overview.	179
Fields	180
Initial values and RPG declaration	182
RPG declaration	183
Chapter 19. MQTMC2 - Trigger message (character format)	185
Overview.	185
Fields	186
Initial values and RPG declaration	187
RPG declaration	187
Chapter 20. MQWIH - Work information header	189
Overview.	189
Fields	189
Initial values and RPG declaration	191
RPG declaration	192
Chapter 21. MQXQH - Transmission queue header	193
Overview.	193
Fields	196
Initial values and RPG declaration	197
RPG declaration	197

Part 2. Function calls 199

Chapter 22. Call descriptions	201
Conventions used in the call descriptions	201
Chapter 23. MQBACK - Back out changes	203
Syntax.	203
Parameters	203
Usage notes	204
RPG invocation.	205
Chapter 24. MQBEGIN - Begin unit of work	207
Syntax.	207
Parameters	207
Usage notes	208
RPG invocation (ILE).	210
Chapter 25. MQCLOSE - Close object	211
Syntax.	211
Parameters	211
Usage notes	214
RPG invocation.	215
Chapter 26. MQCMIT - Commit changes	217
Syntax.	217
Parameters	217
Usage notes	218
RPG invocation.	219
Chapter 27. MQCONN - Connect queue manager	221
Syntax.	221
Parameters	221
Usage notes	224
RPG invocation.	225
Chapter 28. MQCONNX - Connect queue manager (extended)	227
Syntax.	227
Parameters	227
RPG invocation.	228
Chapter 29. MQDISC - Disconnect queue manager	229
Syntax.	229
Parameters	229
Usage notes	230
RPG invocation.	230
Chapter 30. MQGET - Get message	231
Syntax.	231
Parameters	231
Usage notes	235
RPG invocation.	239

Chapter 31. MQINQ - Inquire about object attributes	241
Syntax	241
Parameters	241
Usage notes	248
RPG invocation	250

Chapter 32. MQOPEN - Open object	251
Syntax	251
Parameters	251
Usage notes	259
RPG invocation	264

Chapter 33. MQPUT - Put message	265
Syntax	265
Parameters	265
Usage notes	270
RPG invocation	273

Chapter 34. MQPUT1 - Put one message	275
Syntax	275
Parameters	275
Usage notes	279
RPG invocation	281

Chapter 35. MQSET - Set object attributes	283
Syntax	283
Parameters	283
Usage notes	286
RPG invocation	287

Part 3. Attributes of objects 289

Chapter 36. Attributes of MQSeries objects	291
---	------------

Chapter 37. Attributes for all queues	293
--	------------

Chapter 38. Attributes for local queues and model queues.	299
--	------------

Chapter 39. Attributes for local definitions of remote queues.	313
---	------------

Chapter 40. Attributes for alias queues	315
--	------------

Chapter 41. Attributes for namelists	317
---	------------

Chapter 42. Attributes for process definitions.	319
--	------------

Chapter 43. Attributes for the queue manager	323
---	------------

Part 4. Applications 335

Chapter 44. Building your application	337
MQSeries copy files	337
Preparing your programs to run	337
Interfaces to the AS/400 external syncpoint manager	338
Syncpoints in CICS for AS/400 applications	339

Chapter 45. Sample programs	341
Features demonstrated in the sample programs	342
Preparing and running the sample programs	343
Running the sample programs	343
The Put sample program	343
Design of the Put sample program	344
The Browse sample program	344
Design of the Browse sample program	344
The Get sample program	345
Design of the Get sample program	345
The Request sample program	346
Using triggering with the Request sample	346
Design of the Request sample program	347
The Echo sample program	348
Design of the Echo sample program	349
The Inquire sample program	349
Design of the Inquire sample program	350
The Set sample program	351
Design of the Set sample program	351
The Triggering sample programs	352
The AMQ3TRG4 sample trigger monitor	352
The AMQ3SRV4 sample trigger server	352
Ending the Triggering sample programs	353
Running the samples using remote queues	353

Part 5. Appendixes 355

Appendix A. Return codes	357
Completion codes	357
Reason codes	357

Appendix B. MQSeries constants	421
List of constants	421
LN* (Lengths of character string and byte fields)	421
AC* (Accounting token)	422
ATT* (Accounting token type)	422
AT* (Application type)	423
BND* (Binding)	423
BO* (Begin options)	423
BO* (Begin options structure identifier)	423
BO* (Begin options version)	423
CA* (Character attribute selector)	424
AD* (CICS header ADS descriptor)	424
CC* (Completion code)	425
CS* (Coded character set identifier)	425
CT* (CICS header conversational task)	425
FC* (CICS header facility)	425

CF* (CICS header function name)	425	ME* (Message descriptor extension structure identifier).	436
WI* (CICS header get-wait interval)	425	ME* (Message descriptor extension version)	436
CI* (Correlation identifier)	425	MEF* (Message descriptor extension flags)	437
MQ* (Call identifier)	426	MS* (Message delivery sequence).	437
CIF* (CICS header flags).	426	MF* (Message flags)	437
CI* (CICS header length)	426	MF* (Message-flags masks)	437
CI* (CICS header structure identifier)	426	MI* (Message identifier)	437
CI* (CICS header version)	426	MO* (Match options)	437
LT* (CICS header link type)	426	MT* (Message type)	438
CMLV* (Command level)	427	MTK* (Message token)	438
CN* (Connect options)	427	NC* (Name count)	438
CN* (Connect options structure identifier).	427	OD* (Object descriptor length)	438
CN* (Connect options version)	427	OD* (Object descriptor structure identifier)	438
CO* (Close options)	427	OD* (Object descriptor version)	438
OL* (CICS header output data length)	427	OII* (Object instance identifier)	438
CRC* (CICS header return code)	427	OL* (Original length).	439
SC* (CICS header transaction start code)	428	OO* (Open options)	439
TE* (CICS header task end status)	428	OT* (Object type)	439
CU* (CICS header unit-of-work control)	428	PE* (Persistence)	439
DCC* (Convert-characters masks and factors)	428	PL* (Platform)	439
DCC* (Convert-characters options)	428	PM* (Put message options)	440
DH* (Distribution header structure identifier)	429	PM* (Put message options structure length)	440
DH* (Distribution header version)	429	PM* (Put message options structure identifier)	440
DHF* (Distribution header flags)	429	PM* (Put message options version)	440
DL* (Distribution list support).	429	PF* (Put message record field flags)	440
DL* (Dead-letter header structure identifier)	429	PR* (Priority)	441
DL* (Dead-letter header version)	429	QA* (Inhibit get)	441
DX* (Data-conversion-exit parameter structure identifier).	430	QA* (Inhibit put)	441
DX* (Data-conversion-exit parameter structure version)	430	QA* (Backout hardening)	441
EI* (Expiry interval)	430	QA* (Queue shareability)	441
EN* (Encoding).	430	QD* (Queue definition type)	441
EN* (Encoding masks)	430	QSIE* (Service interval events)	441
EN* (Encoding for packed-decimal integers)	430	QT* (Queue type)	442
EN* (Encoding for floating-point numbers)	430	RC* (Reason code).	442
EN* (Encoding for binary integers)	431	RL* (Returned length)	446
EV* (Event reporting)	431	RM* (Reference message header structure identifier).	446
FB* (Feedback)	431	RM* (Reference message header version)	446
FM* (Format)	432	RM* (Reference message header flags)	446
GI* (Group identifier)	432	RO* (Report options)	446
GM* (Get message options).	432	RO* (Report-options masks)	447
GM* (Get message options structure identifier)	433	SCO* (Queue scope)	447
GM* (Get message options version)	433	SEG* (Segmentation)	447
GS* (Group status)	433	SI* (Security identifier)	447
HC* (Connection handle)	433	SIT* (Security identifier type)	447
HO* (Object handle)	433	SP* (Syncpoint).	448
IA* (Integer attribute selector)	433	SS* (Segment status)	448
IAU* (IMS authenticator)	435	TC* (Trigger control)	448
IAV* (Integer attribute value)	435	TM* (Trigger message structure identifier).	448
ICM* (IMS commit mode)	435	TM* (Trigger message version)	448
II* (IMS header flags).	435	TC* (Trigger message character format structure identifier).	448
II* (IMS header length)	435	TC* (Trigger message character format version)	448
II* (IMS header structure identifier)	435	TT* (Trigger type)	449
II* (IMS header version)	435	US* (Usage)	449
ISS* (IMS security scope)	435	WI* (Wait interval)	449
ITI* (IMS transaction instance identifier)	436	WI* (Workload information header flags)	449
ITS* (IMS transaction state).	436	WI* (Workload information header structure length)	449
MD* (Message descriptor structure identifier)	436		
MD* (Message descriptor version)	436		
ME* (Message descriptor extension length)	436		

WI* (Workload information header structure identifier)	449	MQDXP – Data-conversion exit parameter.	468
WI* (Workload information header version)	449	Fields	468
XR* (Data-conversion-exit response)	450	RPG declaration (ILE)	473
XQ* (Transmission queue header structure identifier)	450	MQXCNVC - Convert characters	473
XQ* (Transmission queue header version)	450	Syntax.	473
		Parameters	473
Appendix C. Rules for validating MQI options	451	RPG invocation.	478
MQOPEN call	451	MQCONVX - Data conversion exit	478
MQPUT call	451	Syntax.	479
MQPUT1 call	452	Parameters	479
MQGET call	452	Usage notes	480
MQCLOSE call	452	RPG invocation.	482
		Appendix G. Notices	483
Appendix D. Machine encodings	453	Programming interface information	484
Binary-integer encoding	453	Trademarks	485
Packed-decimal-integer encoding	454	Glossary of terms and abbreviations	487
Floating-point encoding	454	Bibliography.	499
Constructing encodings	455	MQSeries cross-platform publications	499
Analyzing encodings	455	MQSeries platform-specific publications	501
Using arithmetic	455	Softcopy books	502
Summary of machine architecture encodings	456	BookManager format	502
		HTML format	502
Appendix E. Report options	457	Portable Document Format (PDF)	502
Structure of the report field.	457	PostScript format	502
Analyzing the report field	458	Windows Help format	502
Using arithmetic	458	MQSeries information available on the Internet	502
Structure of the message-flags field	459	Related publications	502
		Index	503
Appendix F. Data conversion.	461	Sending your comments to IBM	509
Conversion processing	461		
Processing conventions	462		
Conversion of report messages	467		

Tables

1.	Elementary data types	5	31.	MQPUT options relating to messages in groups and segments of logical messages . . .	153
2.	RPG COPY files	8	32.	Outcome when MQPUT or MQCLOSE call not consistent with group and segment information	155
3.	ILE RPG bound calls supported by each service program	11	33.	Initial values of fields in MQPMO	163
4.	Fields in MQBO	15	34.	Fields in MQPMR	165
5.	Initial values of fields in MQBO.	16	35.	Fields in MQRMH	169
6.	Fields in MQCIH.	17	36.	Initial values of fields in MQRMH	175
7.	Contents of error information fields in MQCIH structure	18	37.	Fields in MQRR.	177
8.	Initial values of fields in MQCIH	28	38.	Initial values of fields in MQRR	177
9.	Fields in MQCNO	31	39.	Fields in MQTM	179
10.	Initial values of fields in MQCNO	35	40.	Initial values of fields in MQTM	182
11.	Fields in MQDH	37	41.	Fields in MQTMC2	185
12.	Initial values of fields in MQDH	41	42.	Initial values of fields in MQTMC2	187
13.	Fields in MQDLH	43	43.	Fields in MQWIH	189
14.	Initial values of fields in MQDLH	49	44.	Initial values of fields in MQWIH.	191
15.	Fields in MQGMO	51	45.	Fields in MQXQH	193
16.	MQGET options relating to messages in groups and segments of logical messages . . .	63	46.	Initial values of fields in MQXQH	197
17.	Outcome when MQGET or MQCLOSE call not consistent with group and segment information	65	47.	Effect of MQCLOSE options on various types of object and queue	212
18.	Initial values of fields in MQGMO	74	48.	Valid MQOPEN options for each queue type	256
19.	Fields in MQIIH	77	49.	Attributes for all queues	293
20.	Initial values of fields in MQIIH	81	50.	Attributes for local and model queues	299
21.	Fields in MQMD.	83	51.	Attributes for local definitions of remote queues.	313
22.	Initial values of fields in MQMD	129	52.	Attributes for namelists	317
23.	Fields in MQMDE	131	53.	Attributes for process definitions	319
24.	Queue-manager action when MQMDE specified on MQPUT or MQPUT1.	132	54.	Attributes for the queue manager.	323
25.	Initial values of fields in MQMDE	135	55.	Names of the sample programs	341
26.	Fields in MQOD	137	56.	Sample programs demonstrating use of the MQI	342
27.	Initial values of fields in MQOD	145	57.	Client/Server sample program details	348
28.	Fields in MQOR	147	58.	Summary of encodings for machine architectures	456
29.	Initial values of fields in MQOR	148	59.	Fields in MQDXP	468
30.	Fields in MQPMO	149			

About this book

MQSeries for AS/400 Version 5 Release 1 is part of the IBM® MQSeries set of products. It provides application programming services on the AS/400 platform 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 for AS/400 programming interface in the RPG programming language.
- Contains information on how to build an executable application.
- Contains descriptions of sample programs.

Notes to users

1. This book describes the MQSeries for AS/400 programming interface only in the RPG-ILE programming language. If you require details of the RPG-OPM programming language, you should refer to the *MQSeries Application Programming Reference (RPG) V4R2* manual.
2. There are two approaches that can be taken when using the MQI from within an RPG program:
 - Dynamic calls to the QMQM program interface.
 - Static Bound Calls to the MQI procedures.

Using bound calls is generally the preferred method, particularly when the program is making repeated calls to the MQI, as it requires less resource.

New functionality is only available through the Static Bound Call interface.

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 will 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 for AS/400, you need to know how to write programs in the RPG programming language.

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

How to use this book

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

The book is divided into parts:

Part 1. Data type descriptions

Describes the data types that the MQI calls use.

Part 2. Function calls

Describes the parameters and return codes for the calls,

Part 3. Attributes of objects

Describes the attributes of MQSeries for AS/400 objects.

Part 4. Applications

Describes how to build MQSeries for AS/400 programs and the design of the sample applications that are provided with MQSeries for AS/400.

Appearance of text in this book

This book uses the following type styles:

MQOPEN

Example of the name of a call

CMPCOD Example of the name of a parameter of a call

MQMD

Example of the name of a data type or structure

OOSETA

Example of the name of a value

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:

CICS The CICS® for AS/400 product

Also, we use the following shortened name for this language compiler:

RPG Means the IBM ILE RPG for OS/400™ compiler

Part 1. Data type descriptions

Chapter 1. Elementary data types.	3	Initial values and RPG declaration.	49
Conventions used in the descriptions of data types	3	RPG declaration.	49
Elementary data types	3	Chapter 8. MQGMO - Get-message options	51
MQBYTE - Byte	3	Overview	51
MQBYTEN - String of n bytes	4	Fields	51
MQCHAR - character	4	Initial values and RPG declaration.	74
MQCHARn - String of n characters	4	RPG declaration.	74
MQHCONN - Connection handle	4	Chapter 9. MQIIH - IMS bridge header	77
MQHOBJ - Object handle	5	Overview	77
MQLONG - Long integer	5	Fields	78
Elementary data types	5	Initial values and RPG declaration.	81
Chapter 2. Structure data types – programming considerations	7	RPG declaration.	82
Conventions used in the descriptions of data types	7	Chapter 10. MQMD - Message descriptor.	83
Language considerations	8	Overview	84
COPY files	8	Fields	85
Calls	9	Initial values and RPG declaration	129
Structures	10	RPG declaration	130
Notational conventions	10	Chapter 11. MQMDE - Message descriptor extension	131
MQI procedures	10	Overview.	131
Threading considerations.	10	Fields	133
MQI call parameters	11	Initial values and RPG declaration	135
Named constants	11	RPG declaration	136
Commitment control	11	Chapter 12. MQOD - Object descriptor	137
Coding the bound calls	11	Overview.	137
Coding the dynamic calls.	12	Fields	138
Chapter 3. MQBO - Begin options	15	Initial values and RPG declaration	145
Overview	15	RPG declaration	146
Fields	15	Chapter 13. MQOR - Object record	147
Initial values and RPG declaration.	16	Overview.	147
RPG declaration.	16	Fields	147
Chapter 4. MQCIH - CICS bridge header	17	Initial values and RPG declaration	148
Overview	18	RPG declaration	148
Fields	19	Chapter 14. MQPMO - Put message options	149
Initial values and RPG declaration.	28	Overview.	149
RPG declaration.	29	Fields	150
Chapter 5. MQCNO - Connect options.	31	Initial values and RPG declaration	163
Overview	31	RPG declaration	164
Fields	31	Chapter 15. MQPMR - Put-message record	165
Initial values and RPG declaration.	35	Overview.	165
RPG declaration.	35	Fields	165
Chapter 6. MQDH - Distribution header	37	Initial values and RPG declaration	167
Overview	37	RPG declaration	167
Fields	38	Chapter 16. MQRMH - Message reference header	169
Initial values and RPG declaration.	41	Overview.	169
RPG declaration.	41		
Chapter 7. MQDLH - Dead-letter header	43		
Overview	43		
Fields	45		

Data types

Fields	170
Initial values and RPG declaration	175
RPG declaration	176
Chapter 17. MQRR - Response record	177
Overview.	177
Fields	177
Initial values and RPG declaration	177
RPG declaration	178
Chapter 18. MQTM - Trigger message	179
Overview.	179
Fields	180
Initial values and RPG declaration	182
RPG declaration	183
Chapter 19. MQTMC2 - Trigger message (character format)	185
Overview.	185
Fields	186
Initial values and RPG declaration	187
RPG declaration	187
Chapter 20. MQWIH - Work information header	189
Overview.	189
Fields	189
Initial values and RPG declaration	191
RPG declaration	192
Chapter 21. MQXQH - Transmission queue header	193
Overview.	193
Fields	196
Initial values and RPG declaration	197
RPG declaration	197

Chapter 1. Elementary data types

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 a typical declarations in the ILE version of the RPG programming language. The definitions of elementary data types are included here to provide consistency. RPG uses 'D' specifications where working fields can be declared using whatever attributes you need. You can, however, do this in the calculation specifications where the field is used.

To use the elementary data types, you create:

- A /COPY member containing all the data types, or
- An external data structure (PF) containing all the data types. You then need to specify your working fields with attributes 'LIKE' the appropriate data type field.

The benefits of the second option are that the definitions can be used as a 'FIELD REFERENCE FILE' for other AS/400 objects. If an MQ data type definition changes, it is a relatively simple matter to recreate these objects.

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.

Elementary data types

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 “LN* (Lengths of character string and byte fields)” on page 421.

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 326). 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, 20, 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 “LN* (Lengths of character string and byte fields)” on page 421.

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, aligned on its natural boundary.

Elementary data types

Table 1. Elementary data types

Data type	Representation
MQBYTE	A 1-byte alphanumeric field.
MQBYTE16	A 16-byte alphanumeric field.
MQBYTE24	A 24-byte alphanumeric field.
MQBYTE32	A 32-byte alphanumeric field.
MQBYTE64	A 64-byte alphanumeric field.
MQCHAR	A 1-byte alphanumeric field.
MQCHAR4	A 4-byte alphanumeric field.
MQCHAR8	An 8-byte alphanumeric field.
MQCHAR12	A 12-byte alphanumeric field.
MQCHAR16	A 16-byte alphanumeric field.
MQCHAR20	A 20-byte alphanumeric field.
MQCHAR28	A 28-byte alphanumeric field.
MQCHAR32	A 32-byte alphanumeric field.
MQCHAR48	A 48-byte alphanumeric field.
MQCHAR64	A 64-byte alphanumeric field.
MQCHAR128	A 128-byte alphanumeric field.
MQCHAR256	A 256-byte alphanumeric field.
MQHCONN	A 10-digit signed integer.
MQHOBJ	A 10-digit signed integer.
MQLONG	A 10-digit signed integer.
PMQLONG	A 10-digit signed integer.

Elementary data types

Chapter 2. Structure data types – programming considerations

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

- 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
- MQRMH – Message reference header
- 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
- MQDH – Distribution header
- MQDLH – Dead-letter (undelivered-message) header
- MQIIH – IMS™ bridge header
- MQTM – Trigger message
- MQTMC2 – Trigger message (character format 2)
- MQWIH – Work Information header
- MQXQH – Transmission queue header

Note: The MQDXP – data conversion exit parameter structure is in “Appendix F. Data conversion” on page 461, together with the associated data conversion calls.

Conventions used in the descriptions of data types

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 (10-digit signed integer)

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, GMSIDV. A set of constants having the same prefix is shown using the * character, for example: IA*.

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

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

Structure data types

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.

ILE declaration

Typical declaration of the structure in ILE.

Language considerations

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

COPY files

Various COPY files are provided as part of the definition of the message queue interface (MQI), to assist with the writing of RPG application programs that use message queuing. There are two sets of COPY files:

- COPY files with names ending with the letter “G” are for use with programs that use static linkage.
- COPY files with names ending with the letter “R” are for use with programs that use dynamic linkage.

The COPY files reside in QRPGLSRC in the QMQM library.

For each set of COPY files, there are two files containing named constants, and one file for each of the structures. The COPY files are summarized in Table 2.

Table 2. RPG COPY files

Filename (static linkage)	Filename (dynamic linkage)	Contents
CMQBOG	–	Begin options structure
CMQCIHG	CMQCIHR	CICS information header structure
CMQCNOG	–	Connect options structure
CMQDHG	CMQDHR	Distribution header structure
CMQDLHG	CMQDLHR	Dead-letter (undelivered-message) header structure
CMQDXPG	CMQDXPR	Data-conversion-exit parameter structure
CMQGMOG	CMQGMOR	Get-message options structure
CMQIIHG	CMQIIHR	IMS information header structure
CMQMDG	CMQMDR	Message descriptor structure
CMQMDEG	CMQMDER	Message descriptor extension structure
CMQMD1G	CMQMD1R	Message descriptor structure version 1
CMQODG	CMQODR	Object descriptor structure
CMQORG	CMQORR	Object record structure

Table 2. RPG COPY files (continued)

Filename (static linkage)	Filename (dynamic linkage)	Contents
CMQPMOG	CMQPMOR	Put-message options structure
CMQRRG	CMQRRR	Response record structure
CMQTMG	CMQTMR	Trigger-message structure
CMQTMCG	CMQTMCR	Trigger message structure (character format)
CMQTMCG2G	CMQTMCR2R	Trigger message structure (character format) version 2
CMQWIHG	CMQWIHR	Work information header structure
CMQXQHG	CMQXQHR	Transmission-queue header structure
CMQG	CMQR	Named constants for main MQI
CMQXG	CMQXR	Named constants for data-conversion exit

Calls

In “Chapter 22. Call descriptions” on page 201, the calls are described using their individual names. In RPG using dynamic linkage, all calls are made to the single name **QMQM**, and the particular function required is specified by coding an additional parameter which precedes the normal parameters for that call. The following named constants may be used for this additional parameter, in order to identify the function required:

Named constant

Function required

MQCLOS

Close object.

MQCONN

Connect queue manager.

MQDISC

Disconnect queue manager.

MQGET

Get message.

MQINQ

Inquire about object attributes.

MQOPEN

Open object.

MQPUT

Put message.

MQPUT1

Put one message.

MQSET

Set object attributes.

These constants have names which are the same as the calls they identify, with the exception of the constant for the MQCLOSE call, which is abbreviated to MQCLOS.

Note: The calls MQBACK, MQCMIT, and MQCONNX, are not available to applications running in compatibility mode.

Language considerations

Structures

With the exception of the MQTMC structure, all MQ structures are defined with initial values for the fields. These initial values are defined in the relevant table for each structure.

The structure declarations do not contain **DS** statements. This allows the application to declare either a single data structure or a multiple-occurrence data structure, by coding the **DS** statement and then using the **/COPY** statement to copy in the remainder of the declaration:

```
D*..1.....2.....3.....4.....5.....6.....7
D* Declare an MQMD data structure with 5 occurrences
DMYMD          DS                    5
D/COPY CMQMDR
```

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 arrays or character strings whose size is not fixed. For these, a lower case “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.

MQI procedures

When using the ILE bound calls, you must bind to the MQI procedures when you create your program. These procedures are exported from the following service programs as appropriate:

QMQM/AMQZSTUB

This service program provides compatibility bindings for applications written prior to version 5.1 that do not require access to any of the new capabilities provided in version 5.1. The signature of this service program matches that contained in version 4.2.1.

QMQM/LIBMQM

This service program contains the single-threaded bindings for version 5.1. See below for special considerations when writing threaded applications.

QMQM/LIBMQM_R

This service program contains the multi-threaded bindings for version 5.1. See below for special considerations when writing threaded applications.

Use the CRTPGM command to create your programs. For example, the following command would create a single-threaded program that uses the ILE bound calls:

```
CRTPGM PGM(MYPROGRAM) BNDSRVPGM(QMQM/LIBMQM)
```

Threading considerations

In general, RPG programs should not use the multi-threaded service programs. Exceptions are RPG programs created using the version 4.4 ILE RPG compiler and containing the **THREAD(*SERIALIZE)** keyword in the control specification. However, even though these programs are thread-safe, careful consideration must be given to the overall application design, as **THREAD(*SERIALIZE)** forces serialization of RPG procedures at the module level, and this may have an adverse affect on overall performance.

Where RPG programs are used as data-conversion exits, they must be made thread-safe, and should be recompiled using the version 4.4 ILE RPG compiler with THREAD(*SERIALIZE) specified in the control specification.

For further information about threading, see the *AS/400 ILE RPG/400 Reference*, and the *AS/400 ILE RPG/400 Programmer's Guide*.

MQI call parameters

Many parameters passed to the MQI can have more than one concurrent function. This is because the integer value passed is often tested on the setting of individual bits within the field, and not on its total value. This allows you to 'add' several functions together and pass them as a single parameter.

Named constants

There are a large number of different integer and character values that provide data interchange between your application program and the MQI. To facilitate a more readable and consistent approach to using these values, they have all been allocated named constants.

You are recommended to use these named constants and not the values they represent, as this improves the readability of the program source code. Also, if the value of any of these constants should change, you will only need to recompile your program to incorporate the changes.

All named constants are available by referencing the COPY members.

Commitment control

The MQI syncpoint functions MQCMIT and MQBACK are available to ILE RPG programs running in normal mode; these calls allow the program to commit and back out changes to MQ resources.

The MQCMIT and MQBACK calls are not available to ILE RPG programs running in compatibility mode. For these programs you should use the operation codes COMMIT and ROLBK.

Coding the bound calls

MQI ILE procedures are listed in Table 3.

Table 3. ILE RPG bound calls supported by each service program

Name of call	LIBMQM and LIBMQM_R	AMQZSTUB	AMQVSTUB
MQBACK	✓		
MQBEGIN	✓		
MQCMIT	✓		
MQCLOSE	✓	✓	
MQCONN	✓	✓	
MQCONNX	✓		
MQDISC	✓	✓	
MQGET	✓	✓	
MQINQ	✓	✓	

Language considerations

Table 3. ILE RPG bound calls supported by each service program (continued)

Name of call	LIBMQM and LIBMQM_R	AMQZSTUB	AMQVSTUB
MQOPEN	✓	✓	
MQPUT	✓	✓	
MQPUT1	✓	✓	
MQSET	✓	✓	
MQXCNVC	✓		✓

To use these procedures you need to:

1. Define the external procedures in your 'D' specifications. These are all available within the COPY file member CMQG containing the named constants.
2. Use the CALLP operation code to call the procedure along with its parameters.

For example the MQOPEN call requires the inclusion of the following code:

```

D*****
D** MQOPEN Call -- Open Object (From COPY file CMQG) **
D*****
D*
D*.1.....2.....3.....4.....5.....6.....7..
DMQOPEN PR EXTPROC('MQOPEN')
D* Connection handle
D HCONN 10I 0 VALUE
D* Object descriptor
D OBJDSC 224A
D* Options that control the action of MQOPEN
D OPTS 10I 0 VALUE
D* Object handle
D HOBJ 10I 0
D* Completion code
D CMPCOD 10I 0
D* Reason code qualifying CMPCOD
D REASON 10I 0
D*

```

To call the procedure, after initializing the various parameters, you need the following code:

```

...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...8
C CALLP MQOPEN(HCONN : MQOD : OPTS : HOBJ :
C CMPCOD : REASON)

```

Here, the structure MQOD is defined using the COPY member CMQODG which breaks it down into its components.

Coding the dynamic calls

To use the MQI through dynamic calls to QMQM, you require the following code. The example is again MQOPEN:

```

...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...8
C Z-ADD MQOPEN CID
C CALL 'QMQM'
C PARM CID 9 0
C PARM HCONN 9 0
C PARM MQOD
C PARM OPTS 9 0
C PARM HOBJ 9 0
C PARM CMPCOD 9 0
C PARM REASON 9 0
*
```

Language considerations

Here, the structure MQOD is defined using the COPY member CMQODR which splits it into its components.

Note: Function that is new in version 5 release 1 cannot be accessed by calls that use dynamic linkage. For example, the MQBACK, MQBEGIN, MQCMIT, and MQCONNX calls can be used only with static linkage.

Language considerations

Chapter 3. MQBO - Begin options

The following table summarizes the fields in the structure.

Table 4. Fields in MQBO

Field	Description	Page
<i>BOSID</i>	Structure identifier	15
<i>BOVER</i>	Structure version number	15
<i>BOOPT</i>	Options that control the action of MQBEGIN	15

Overview

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, OS/400, Sun Solaris, Windows NT.

Fields

BOSID (4-byte character string)
Structure identifier.

The value must be:

BOSIDV

Identifier for begin-options structure.

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

BOVER (10-digit signed integer)
Structure version number.

The value must be:

BOVER1

Version number for begin-options structure.

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

BOVERC

Current version of begin-options structure.

This is always an input field. The initial value of this field is BOVER1.

BOOPT (10-digit signed integer)
Options that control the action of MQBEGIN.

The value must be:

BONONE

No options specified.

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

Initial values and RPG declaration

Table 5. Initial values of fields in MQBO

Field name	Name of constant	Value of constant
<i>BOSID</i>	BOSIDV	'B0bb' (See note 1)
<i>BOVER</i>	BOVER1	1
<i>BOOPT</i>	BONONE	0
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*.1.....2.....3.....4.....5.....6.....7..
D* MQBO Structure
D*
D* Structure identifier
D BOSID          1      4
D* Structure version number
D BOVER          5      8I 0
D* Options that control the action of MQBEGIN
D BOOPT          9      12I 0
    
```

Chapter 4. MQCIH - CICS bridge header

The following table summarizes the fields in the structure.

Table 6. Fields in MQCIH

Field	Description	Page
<i>CISID</i>	Structure identifier	19
<i>CIVER</i>	Structure version number	19
<i>CILEN</i>	Length of MQCIH structure	19
<i>CIENC</i>	Reserved	19
<i>CICSI</i>	Reserved	19
<i>CIFMT</i>	MQ™ format name of data that follows MQCIH	20
<i>CIFLG</i>	Flags	20
<i>CIRET</i>	Return code from bridge	20
<i>CICC</i>	MQ completion code or CICS EIBRESP	21
<i>CIREA</i>	MQ reason or feedback code, or CICS EIBRESP2	21
<i>CIUOW</i>	Unit-of-work control	21
<i>CIGWI</i>	Wait interval for MQGET call issued by bridge task	21
<i>CILT</i>	Link type	22
<i>CIODL</i>	Output COMMAREA data length	22
<i>CIFKT</i>	Bridge facility release time	22
<i>CIADS</i>	Send/receive ADS descriptor	23
<i>CICT</i>	Whether task can be conversational	23
<i>CITES</i>	Status at end of task	23
<i>CIFAC</i>	BVT token value	24
<i>CIFNC</i>	MQ call name or CICS EIBFN function	24
<i>CIAC</i>	Abend code	24
<i>CIAUT</i>	Password or passticket	25
<i>CIRFM</i>	MQ format name of reply message	25
<i>CIRSI</i>	Remote sysid to use	25
<i>CIRTI</i>	Remote transid to attach	25
<i>CITI</i>	Transaction to attach	25
<i>CIFL</i>	Terminal emulated attributes	26
<i>CIAI</i>	AID key	26
<i>CISC</i>	Transaction start code	26
<i>CICNC</i>	Abend transaction code	26
<i>CINTI</i>	Next transaction to attach	27
Note: The remaining fields are not present if <i>CIVER</i> is less than <i>CIVER2</i> .		
<i>CICP</i>	Cursor position	27
<i>CIEO</i>	Offset of error in message	27

MQCIH - CICS bridge header

Table 6. Fields in MQCIH (continued)

Field	Description	Page
<i>CIII</i>	Item number of last message read	27

Overview

The MQCIH structure describes the information that can be present at the start of a message sent to the CICS bridge through MQSeries for OS/390™. The structure can be omitted if the values required by the application are the same as the initial values shown in Table 8 on page 28 and the bridge is running with AUTH=LOCAL or IDENTIFY. The format name of this structure is FMCICS.

The current version of MQCIH is CIVER2. Fields that exist only in the version-2 structure are identified as such in the descriptions that follow. The declaration of MQCIH provided in the COPY file contains the new fields, with the initial value of the *CIVER* field set to GMVER2.

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

The application must ensure that fields documented as “request” fields have appropriate values in the message that the application sends to the CICS bridge; these fields are input to the bridge. Fields documented as “response” fields are set by the CICS bridge in the reply message that the bridge sends to the application.

Error information is returned in the *CIRET*, *CIFNC*, *CICC*, *CIREA*, and *CIAC* fields. Which of them is set depends on the value of the *CIRET* field; see Table 7.

Table 7. Contents of error information fields in MQCIH structure

<i>CIRET</i>	<i>CIFNC</i>	<i>CICC</i>	<i>CIREA</i>	<i>CIAC</i>
CRC000	-	-	-	-
CRC003	-	-	FBC*	-
CRC002 CRC008	MQ call name	MQ <i>CMPCOD</i>	MQ <i>REASON</i>	-
CRC001 CRC006 CRC007 CRC009	CICS EIBFN	CICS EIBRESP	CICS EIBRESP2	-
CRC004 CRC005	-	-	-	CICS ABCODE

Fields

CISID (4-byte character string)
Structure identifier.

The value must be:

CISIDV

Identifier for CICS information header structure.

This is a request field. The initial value of this field is CISIDV.

CIVER (10-digit signed integer)
Structure version number.

The value must be one of the following:

CIVER1

Version-1 CICS information header structure.

CIVER2

Version-2 CICS information header structure.

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:

CIVERC

Current version of CICS information header structure.

This is a request field. The initial value of this field is CIVERC.

CILEN (10-digit signed integer)
Length of MQCIH structure.

The value must be one of the following:

CILEN1

Length of version-1 CICS information header structure.

CILEN2

Length of version-2 CICS information header structure.

The following constant specifies the length of the current version:

CILENC

Length of current version of CICS information header structure.

This is a request field. The initial value of this field is CILEN2.

CIENC (10-digit signed integer)
Reserved.

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

CICSI (10-digit signed integer)
Reserved.

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

MQCIH - CICS bridge header

CIFMT (8-byte character string)

MQ format name of data that follows MQCIH.

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

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

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

This is a request field. The length of this field is given by LNFMT. The initial value of this field is FMNONE.

CIFLG (10-digit signed integer)

Flags.

The value must be:

CIFNON

No flags.

This is a request field. The initial value of this field is CIFNON.

CIRET (10-digit signed integer)

Return code from bridge.

This is the return code from the CICS bridge describing the outcome of the processing performed by the bridge. The *CIFNC*, *CICC*, *CIREA*, and *CIAC* fields may contain additional information (see Table 7 on page 18). The value is one of the following:

CRC000

(0, X'000') No error.

CRC001

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

CRC002

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

CRC003

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

CRC004

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

CRC005

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

CRC006

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

CRC007

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

CRC008

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

CRC009

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

This is a response field. The initial value of this field is CRC000.

CICC (10-digit signed integer)

MQ completion code or CICS EIBRESP.

The value returned in this field is dependent on *CIRET*; see Table 7 on page 18.

This is a response field. The initial value of this field is CCOK

CIREA (10-digit signed integer)

MQ reason or feedback code, or CICS EIBRESP2.

The value returned in this field is dependent on *CIRET*; see Table 7 on page 18.

This is a response field. The initial value of this field is RCNONE.

CIUOW (10-digit signed integer)

Unit-of-work control.

This controls the unit-of-work processing performed by the CICS bridge. You can request the bridge to run a single transaction, or one or more programs within a unit of work. 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:

CUONLY

Start unit of work, perform function, then commit the unit of work (DPL and 3270).

CUCONT

Additional data for the current unit of work (3270 only).

CUFRST

Start unit of work and perform function (DPL only).

CUMIDL

Perform function within current unit of work (DPL only).

CULAST

Perform function, then commit the unit of work (DPL only).

CUCMIT

Commit the unit of work (DPL only).

CUBACK

Back out the unit of work (DPL only).

This is a request field. The initial value of this field is CUONLY.

CIGWI (10-digit signed integer)

Wait interval for MQGET call issued by bridge task.

This field is applicable only when *CIUOW* has the value CUFRST. 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

MQCIH - CICS bridge header

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:

WIDFLT

Default wait interval.

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

WIULIM

Unlimited wait interval.

This is a request field. The initial value of this field is WIDFLT.

CILT (10-digit signed integer)

Link type.

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

LTPROG

DPL program.

LTTRAN

3270 transaction.

This is a request field. The initial value of this field is LTPROG.

CIODL (10-digit signed integer)

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 *CIODL*, 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 can be used:

OLINPT

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.

This is a request field used only for DPL programs. The initial value of this field OLINPT.

CIFKT (10-digit signed integer)

Bridge facility release time.

This is the length of time in seconds that the bridge facility will be kept after the user transaction has ended.

This is a request field used only for 3270 transactions. The initial value of this field is 0.

CIADS (10-digit signed integer)

Send/receive ADS descriptor.

This is an indicator specifying whether ADS descriptors should be sent on SEND and RECEIVE BMS requests. The value must be one of the following:

ADNONE

No ADS descriptor.

ADSEND

Send ADS descriptor.

ADRECV

Receive ADS descriptor.

ADMSGF

Receive ADS descriptor.

This is a request field used only for 3270 transactions. The initial value of this field is ADNONE.

CICT (10-digit signed integer)

Whether task can be conversational.

This is an indicator specifying whether the task should be allowed to issue requests for more information, or should abend. The value must be one of the following:

CTYES

Task is conversational.

CTNO

Task is not conversational.

This is a request field used only for 3270 transactions. The initial value of this field is CTNO.

CITES (10-digit signed integer)

Status at end of task.

This field shows the status of the user transaction at end of task. One of the following values is returned:

TENOSY

Not synchronized.

The user transaction has not yet completed and has not syncpointed.

TECMIT

Commit unit of work.

The user transaction has not yet completed, but has syncpointed the first unit of work.

TEBACK

Back out unit of work.

The user transaction has not yet completed. The current unit of work will be backed out.

MQCIH - CICS bridge header

TEENDT

End task.

The user transaction has ended (or abended).

This is a response field used only for 3270 transactions. The initial value of this field is TENOSY.

CIFAC (8-byte bit string)

BVT token value.

This is an 8-byte bridge-facility token. The purpose of a bridge-facility token is to allow multiple transactions in a pseudoconversation to use the same bridge facility (virtual 3270 terminal). In the first, or only, message in a pseudoconversation, a value of FCNONE should be set; this tells CICS to allocate a new bridge facility for this message. A bridge-facility token is returned in response messages when a nonzero *CIFKT* is specified on the input message. Subsequent input messages can then use the same bridge-facility token.

The following special value is defined:

FCNONE

No BVT token specified.

This is both a request and a response field used only for 3270 transactions. The length of this field is given by LNFAC. The initial value of this field is FCNONE.

CIFNC (4-byte character string)

MQ call name or CICS EIBFN function.

The value returned in this field is dependent on *CIRET*; see Table 7 on page 18. The following values are possible when *CIFNC* contains an MQ call name:

CFCONN

MQCONN call.

CFGET

MQGET call.

CFINQ

MQINQ call.

CFOPEN

MQOPEN call.

CFPUT

MQPUT call.

CFPUT1

MQPUT1 call.

CFNONE

No call.

This is a response field. The length of this field is given by LNFUNC. The initial value of this field is CFNONE.

CIAC (4-byte character string)

Abend code.

MQCIH - CICS bridge header

The value returned in this field is dependent on *CIRET*; see Table 7 on page 18.

This is a response field. The length of this field is given by *LNABNC*. The initial value of this field is 4 blank characters.

CIAUT (8-byte character string)

Password or passticket.

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

This is a request field. The length of this field is given by *LNAUTH*. The initial value of this field is 8 blanks.

CIRSI (8-byte character string)

Reserved.

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

CIRFM (8-byte character string)

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 *MDFMT* field in MQMD.

This is a request field used only for DPL programs. The length of this field is given by *LNFMT*. The initial value of this field is FMNONE.

CIRSI (4-byte character string)

Remote sysid to use.

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

CIRTI (4-byte character string)

Remote transid to attach.

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

CITI (4-byte character string)

Transaction to attach.

If *CILT* has the value *LTTRAN*, *CITI* is the transaction identifier of the user transaction to be run; a nonblank value must be specified in this case.

If *CILT* has the value *LTPROG*, *CITI* 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 DPL bridge default transaction code (*CKBP*) is used. If the value is nonblank, it must have been defined to CICS as a local TRANSACTION whose initial program is *CSQCBP00*. This field is applicable only when *CIUOW* has the value *CUFRST* or *CUONLY*.

This is a request field. The length of this field is given by *LNTRID*. The initial value of this field is 4 blanks.

MQCIH - CICS bridge header

CIFL (4-byte character string)
Terminal emulated attributes.

This is the name of an installed terminal that is to be used as a model for the bridge facility. A value of blanks means that *CIFL* is taken from the bridge transaction profile definition, or a default value is used.

This is a request field used only for 3270 transactions. The length of this field is given by *LNFACL*. The initial value of this field is 4 blanks.

CIAT (4-byte character string)
AID key.

This is the initial value of the AID key when the transaction is started. It is a 1-byte value, left justified.

This is a request field used only for 3270 transactions. The length of this field is given by *LNATID*. The initial value of this field is 4 blanks.

CISC (4-byte character string)
Transaction start code.

This is an indicator specifying whether the bridge emulates a terminal transaction or a STARTed transaction. The value must be one of the following:

SCSTRT
Start.

SCDATA
Start data.

SCTERM
Terminate input.

SCNONE
None.

In the response from the bridge, this field is set to the start code appropriate to the next transaction ID contained in the *CINTI* field. The following start codes are possible in the response:

SCSTRT
SCDATA
SCTERM

For CICS Transaction Server Version 1.2, this field is a request field only; its value in the response is undefined.

For CICS Transaction Server Version 1.3 and subsequent releases, this is both a request and a response field.

This field is used only for 3270 transactions. The length of this field is given by *LNSTCO*. The initial value of this field is *SCNONE*.

CICNC (4-byte character string)
Abend transaction code.

This is the abend code to be used to terminate the transaction (normally a conversational transaction that is requesting more data). Otherwise this field is set to blanks.

MQCIH - CICS bridge header

This is a request field used only for 3270 transactions. The length of this field is given by LNCNCL. The initial value of this field is 4 blanks.

CINTI (4-byte character string)
Next transaction to attach.

This is the name of the next transaction returned by the user transaction (usually by EXEC CICS RETURN TRANSID). If there is no next transaction, this field is set to blanks.

This is a response field used only for 3270 transactions. The length of this field is given by LNTRID. The initial value of this field is 4 blanks.

CIRS2 (8-byte character string)
Reserved.

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

CIRS3 (8-byte character string)
Reserved.

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

The remaining fields are not present if *CIVER* is less than *CIVER2*.

CICP (10-digit signed integer)
Cursor position.

This is the initial cursor position when the transaction is started. Subsequently, for conversational transactions, the cursor position is in the RECEIVE vector.

This is a request field used only for 3270 transactions. The initial value of this field is 0. This field is not present if *CIVER* is less than *CIVER2*.

CIE0 (10-digit signed integer)
Offset of error in message.

This is the position of invalid data detected by the bridge exit. This field provides the offset from the start of the message to the location of the invalid data.

This is a response field used only for 3270 transactions. The initial value of this field is 0. This field is not present if *CIVER* is less than *CIVER2*.

CIII (10-digit signed integer)
Item number of last message read.

This is a reserved field. The value must be 0. This field is not present if *CIVER* is less than *CIVER2*.

CIRS4 (10-digit signed integer)
Reserved.

This is a reserved field. The value must be 0. This field is not present if *CIVER* is less than *CIVER2*.

Initial values and RPG declaration

Table 8. Initial values of fields in MQCIH

Field name	Name of constant	Value of constant
<i>CISID</i>	CISIDV	'CIHb' (See note 1)
<i>CIVER</i>	CIVER2	2
<i>CILEN</i>	CILEN2	180
<i>CIENC</i>	None	0
<i>CICSI</i>	None	0
<i>CIFMT</i>	FMNONE	'bbbbbbbb'
<i>CIFLG</i>	CIFNON	0
<i>CIRET</i>	CRC000	0
<i>CICC</i>	CCOK	0
<i>CIREA</i>	RCNONE	0
<i>CIUOW</i>	CUONLY	273
<i>CIGWI</i>	WIDFLT	-2
<i>CILT</i>	LTPROG	1
<i>CIODL</i>	OLINPT	-1
<i>CIFKT</i>	None	0
<i>CIADS</i>	ADNONE	0
<i>CICT</i>	CTNO	0
<i>CITES</i>	TENOSY	0
<i>CIFAC</i>	FCNONE	Nulls
<i>CIFNC</i>	CFNONE	'bbbb'
<i>CIAC</i>	None	'bbbb'
<i>CIAUT</i>	None	'bbbbbbbb'
<i>CIRS1</i>	None	'bbbbbbbb'
<i>CIRFM</i>	FMNONE	'bbbbbbbb'
<i>CIRSI</i>	None	'bbbb'
<i>CIRTI</i>	None	'bbbb'
<i>CITI</i>	None	'bbbb'
<i>CIFL</i>	None	'bbbb'
<i>CAI</i>	None	'bbbb'
<i>CISC</i>	SCNONE	'bbbb'
<i>CICNC</i>	None	'bbbb'
<i>CINTI</i>	None	'bbbb'
<i>CIRS2</i>	None	'bbbbbbbb'
<i>CIRS3</i>	None	'bbbbbbbb'
<i>CICP</i>	None	0
<i>CIEO</i>	None	0
<i>CIII</i>	None	0
<i>CIRS4</i>	None	0

Table 8. Initial values of fields in MQCIH (continued)

Field name	Name of constant	Value of constant
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQCIH Structure
D*
D* Structure identifier
D  CISID          1      4
D* Structure version number
D  CIVER          5      8I 0
D* Length of MQCIH structure
D  CILEN          9     12I 0
D* Reserved
D  CIENC         13     16I 0
D* Reserved
D  CICSI         17     20I 0
D* MQ format name of data that follows MQCIH
D  CIFMT         21     28
D* Flags
D  CIFLG         29     32I 0
D* Return code from bridge
D  CIRET         33     36I 0
D* MQ completion code or CICS EIBRESP
D  CICC          37     40I 0
D* MQ reason or feedback code, or CICS EIBRESP2
D  CIREA         41     44I 0
D* Unit-of-work control
D  CIUOW         45     48I 0
D* Wait interval for MQGET call issued by bridge task
D  CIGWI         49     52I 0
D* Link type
D  CILT          53     56I 0
D* Output COMMAREA data length
D  CIODL         57     60I 0
D* Bridge facility release time
D  CIFKT         61     64I 0
D* Send/receive ADS descriptor
D  CIADS         65     68I 0
D* Whether task can be conversational
D  CICT          69     72I 0
D* Status at end of task
D  CITES         73     76I 0
D* BVT token value
D  CIFAC         77     84
D* MQ call name or CICS EIBFN function
D  CIFNC         85     88
D* Abend code
D  CIAC          89     92
D* Password or passticket
D  CIAUT         93    100
D* Reserved
D  CIRS1        101    108
D* MQ format name of reply message
D  CIRFM        109    116
D* Remote sysid to use
D  CIRSI        117    120
D* Remote transid to attach
D  CIRTI        121    124
D* Transaction to attach
D  CITI         125    128

```

RPG declaration

```
D* Terminal emulated attributes
D CIFL          129  132
D* AID key
D CIAI          133  136
D* Transaction start code
D CISC          137  140
D* Abend transaction code
D CICNC         141  144
D* Next transaction to attach
D CINTI         145  148
D* Reserved
D CIRS2         149  156
D* Reserved
D CIRS3         157  164
D* Cursor position
D CICP          165  168I 0
D* Offset of error in message
D CIE0          169  172I 0
D* Item number of last message read
D CIII          173  176I 0
D* Reserved
D CIRS4         177  180I 0
```

Chapter 5. MQCNO - Connect options

The following table summarizes the fields in the structure.

Table 9. Fields in MQCNO

Field	Description	Page
<i>CNSID</i>	Structure identifier	31
<i>CNVER</i>	Structure version number	31
<i>CNOPT</i>	Options that control the action of MQCONNX	31
Note: The remaining fields are not present if <i>CNVER</i> is less than <i>CNVER2</i> .		
<i>CNCCO</i>	Offset of MQCD structure for client connection	33
<i>CNCCP</i>	Address of MQCD structure for client connection	33

Overview

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

Fields

CNSID (4-byte character string)
Structure identifier.

The value must be:

CNSIDV

Identifier for connect-options structure.

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

CNVER (10-digit signed integer)
Structure version number.

The value must be one of the following:

CNVER1

Version-1 connect-options structure.

CNVER2

Version-2 connect-options structure.

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:

CNVERC

Current version of connect-options structure.

This is always an input field. The initial value of this field is *CNVER1*.

CNOPT (10-digit signed integer)
Options that control the action of MQCONNX.

MQCNO - Connect options

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

CNSBND

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.

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

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

CNFBND

Fastpath 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.

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

CNFBND 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. 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 fastpath binding:

- **Use of the CNFBND option compromises the integrity of the queue manager, because 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.**
- The application must not use asynchronous signals or timer interrupts (such as `sigkill`) with CNFBND. There are also restrictions on the use of shared memory segments. Refer to the *MQSeries Application Programming Guide* for more information.
- The application must not have more than one thread connected to the queue manager at any one time.
- The application must use the `MQDISC` call to disconnect from the queue manager.
- The application must finish before ending the queue manager with the `endmqm` command.

The following points apply to the use of CNFBND in the environments indicated:

- On OS/400, the job must run under a user profile that belongs to the QMQMADM group. Also, the program must not terminate abnormally, otherwise unpredictable results may occur.

MQCNO - Connect options

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

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

CNNONE

No options specified.

CNNONE 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 CNNONE.

The remaining fields are not present if *CNVER* is less than *CNVER2*.

CNCCO (10-digit signed integer)

Offset of MQCD structure for client connection.

This is the offset in bytes of an MQCD channel definition structure from the start of the MQCNO structure. The offset can be positive or negative.

CNCCO is used only when the application issuing the MQCONN call is running as an MQ client. For information on how to use this field, see the description of the *CNCCP* field.

This is an input field. The initial value of this field is 0. This field is not present if *CNVER* is less than *CNVER2*.

CNCCP (pointer)

Address of MQCD structure for client connection.

CNCCO and *CNCCP* are used only when the application issuing the MQCONN call is running as an MQ client. By specifying one or other of these fields, the application can control the definition of the client connection channel by providing an MQCD channel definition structure that contains the values required.

If the application is running as an MQ client but the application does not provide an MQCD structure, the MQSERVER environment variable is used to select the channel definition. If MQSERVER is not set, the client channel table is used.

If the application is not running as an MQ client, *CNCCO* and *CNCCP* are ignored.

If the application provides an MQCD structure, the fields listed below must be set to the values required; other fields in MQCD are ignored. Character strings can be padded with blanks to the length of the field, or terminated by a null character. Refer to the *MQSeries Intercommunication* book for more information about the fields in the MQCD structure.

Field in MQCD

CDCHN
CDVER
CDTRT
CDMOD
CDTP

Value

Channel name.
Structure version number. Must not be less than *CDVER6*.
Any supported transport type.
LU 6.2 mode name.
LU 6.2 transaction program name.

MQCNO - Connect options

Field in MQCD	Value
<i>CDSCX</i>	Name of channel security exit.
<i>CDSNX</i>	Name of channel send exit.
<i>CDRCX</i>	Name of channel receive exit.
<i>CDMML</i>	Maximum length in bytes of messages that can be sent over the client connection channel.
<i>CDSCD</i>	User data for security exit.
<i>CDSND</i>	User data for send exit.
<i>CDRCD</i>	User data for receive exit.
<i>CDUID</i>	User identifier to be used to establish an LU 6.2 session.
<i>CDPW</i>	Password to be used to establish an LU 6.2 session.
<i>CDCON</i>	Connection name.
<i>CDHBI</i>	Time in seconds between heartbeat flows.
<i>CDLEN</i>	Length of the MQCD structure.
<i>CDXNL</i>	Length of exit names addressed by <i>CDSXP</i> and <i>CDRXP</i> . Must be greater than zero if <i>CDSXP</i> or <i>CDRXP</i> is set to a value that is not the null pointer.
<i>CDXDL</i>	Length of exit data addressed by <i>CDSUP</i> and <i>CDRUP</i> . Must be greater than zero if <i>CDSUP</i> or <i>CDRUP</i> is set to a value that is not the null pointer.
<i>CDSXD</i>	Number of send exits addressed by <i>CDSXP</i> . If zero, <i>CDSNX</i> and <i>CDSND</i> provide the exit name and data. If greater than zero, <i>CDSXP</i> and <i>CDSUP</i> provide the exit names and data, and <i>CDSNX</i> and <i>CDSND</i> must be blank.
<i>CDRXD</i>	Number of receive exits addressed by <i>CDRXP</i> . If zero, <i>CDRCX</i> and <i>CDRCD</i> provide the exit name and data. If greater than zero, <i>CDRXP</i> and <i>CDRUP</i> provide the exit names and data, and <i>CDRCX</i> and <i>CDRCD</i> must be blank.
<i>CDSXP</i>	Address of name of first send exit.
<i>CDSUP</i>	Address of data for first send exit.
<i>CDRXP</i>	Address of name of first receive exit.
<i>CDRUP</i>	Address of data for first receive exit.
<i>CDLRL</i>	Length of long remote user identifier.
<i>CDLRP</i>	Address of long remote user identifier.
<i>CDRSI</i>	Remote security identifier.

The channel definition structure can be provided in one of two ways:

- By using the offset field *CNCCO*
In this case, the application should declare its own structure containing an MQCNO followed by the channel definition structure MQCD, and set *CNCCO* to the offset of the channel definition structure from the start of the MQCNO. Care must be taken to ensure that this offset is correct. *CNCCP* must be set to the null pointer or null bytes.
- By using the pointer field *CNCCP*
In this case, the application can declare the channel definition structure separately from the MQCNO structure, and set *CNCCP* to the address of the channel definition structure. *CNCCO* must be set to zero.

Whichever technique is chosen, only one of *CNCCO* and *CNCCP* can be used; the call fails with reason code RC2278 if both are nonzero.

Once the MQCONN called has completed, the MQCD structure is not referenced again.

This is an input field. The initial value of this field is the null pointer. This field is not present if *CNVER* is less than *CNVER2*.

Initial values and RPG declaration

Table 10. Initial values of fields in MQCNO

Field name	Name of constant	Value of constant
<i>CNSID</i>	CNSIDV	'CNO b ' (See note 1)
<i>CNVER</i>	CNVER1	1
<i>CNOPT</i>	CNNONE	0
<i>CNCCO</i>	None	0
<i>CNCCP</i>	None	Null pointer or null bytes
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQCNO Structure
D*
D* Structure identifier
D CNSID          1      4
D* Structure version number
D CNVER          5      8I 0
D* Options that control the action of MQCONN
D CNOPT          9      12I 0
D* Offset of MQCD structure for client connection
D CNCCO         13      16I 0
D* Address of MQCD structure for client connection
D CNCCP         17      32*

```

MQCNO - Connect options

Chapter 6. MQDH - Distribution header

The following table summarizes the fields in the structure.

Table 11. Fields in MQDH

Field	Description	Page
<i>DHSID</i>	Structure identifier	38
<i>DHVER</i>	Structure version number	38
<i>DHLEN</i>	Length of MQDH structure plus following records	38
<i>DHENC</i>	Numeric encoding of data that follows array of MQPMR records	39
<i>DHCSI</i>	Character set identifier of data that follows array of MQPMR records	39
<i>DHFMT</i>	Format name of data that follows array of MQPMR records	39
<i>DHFLG</i>	General flags	39
<i>DHPRF</i>	Flags indicating which MQPMR fields are present	40
<i>DHCNT</i>	Number of object records present	40
<i>DHORO</i>	Offset of first object record from start of MQDH	40
<i>DHPRO</i>	Offset of first put-message record from start of MQDH	40

Overview

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

MQDH - Distribution header

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 RC2135.

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 “Chapter 38. Attributes for local queues and model queues” on page 299). 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

DHSID (4-byte character string)
Structure identifier.

The value must be:

DHSIDV

Identifier for distribution header structure.

The initial value of this field is DHSIDV.

DHVER (10-digit signed integer)
Structure version number.

The value must be:

DHVER1

Version number for distribution header structure.

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

DHVERC

Current version of distribution header structure.

The initial value of this field is DHVER1.

DHLEN (10-digit signed integer)
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 *DHLEN*, but the content of those bytes is not preserved by the queue manager. It is valid for the array of MQPMPR records to precede the array of MQOR records.

The initial value of this field is 0.

DHENC (10-digit signed integer)

Numeric encoding of data that follows array of MQPMPR records.

The initial value of this field is 0.

DHCSI (10-digit signed integer)

Character set identifier of data that follows array of MQPMPR records.

The initial value of this field is 0.

DHFMT (8-byte character string)

Format name of data that follows array of MQPMPR records.

The initial value of this field is FMNONE.

DHFLG (10-digit signed integer)

General flags.

The following flag can be specified:

DHFNEW

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 *PRMID* 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 DHFNEW 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 *PRMID* field.
- The *MDMID* field in MQMD is MINONE, or the *PMOPT* field in MQPMO includes PMNMID

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

DHFNON

No flags.

This constant indicates that no flags have been specified. DHFNON 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 DHFNON.

MQDH - Distribution header

DHPRF (10-digit signed integer)

Flags indicating which MQPMPR fields are present.

Zero or more of the following flags can be specified:

PFMID

Message-identifier field is present.

PFCID

Correlation-identifier field is present.

PFGID

Group-identifier field is present.

PFFB Feedback field is present.

PFACC

Accounting-token field is present.

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

PFNONE

No put-message record fields are present.

PFNONE 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 PFNONE.

DHCNT (10-digit signed integer)

Number of object records present.

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

The initial value of this field is 0.

DHORO (10-digit signed integer)

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 *DHCNT* 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 *DHLEN* field.

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

The initial value of this field is 0.

DHPRO (10-digit signed integer)

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 MQPMPR put message records containing the message properties. If present, there are *DHCNT* 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 *DHLEN* field.

Put message records are optional; if no records are provided, *DHPRO* is zero, and *DHPRF* has the value PFNONE.

The initial value of this field is 0.

Initial values and RPG declaration

Table 12. Initial values of fields in MQDH

Field name	Name of constant	Value of constant
<i>DHSID</i>	DHSIDV	'DHbb' (See note 1)
<i>DHVER</i>	DHVER1	1
<i>DHLEN</i>	None	0
<i>DHENC</i>	None	0
<i>DHCSI</i>	None	0
<i>DHFMT</i>	FMNONE	'bbbbbbbb'
<i>DHFLG</i>	DHFNON	0
<i>DHPRF</i>	PFNONE	0
<i>DHCNT</i>	None	0
<i>DHORO</i>	None	0
<i>DHPRO</i>	None	0
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQDH Structure
D*
D* Structure identifier
D DHSID          1      4
D* Structure version number
D DHVER          5      8I 0
D* Length of MQDH structure plus following records
D DHLEN          9      12I 0
D* Numeric encoding of data that follows array of MQPMR records
D DHENC         13      16I 0
D* Character set identifier of data that follows array of MQPMR
D* records
D DHCSI         17      20I 0
D* Format name of data that follows array of MQPMR records
D DHFMT         21      28
D* General flags
D DHFLG         29      32I 0
D* Flags indicating which MQPMR fields are present
D DHPRF         33      36I 0
D* Number of object records present
D DHCNT         37      40I 0
D* Offset of first object record from start of MQDH
D DHORO         41      44I 0
D* Offset of first put message record from start of MQDH
D DHPRO         45      48I 0

```

MQDH - Distribution header

Chapter 7. MQDLH - Dead-letter header

The following table summarizes the fields in the structure.

Table 13. Fields in MQDLH

Field	Description	Page
<i>DLSID</i>	Structure identifier	45
<i>DLVER</i>	Structure version number	45
<i>DLREA</i>	Reason message arrived on dead-letter queue	45
<i>DLDQ</i>	Name of original destination queue	46
<i>DLDM</i>	Name of original destination queue manager	46
<i>DLENC</i>	Numeric encoding of data that follows MQDLH	46
<i>DLCSI</i>	Character set identifier of data that follows MQDLH	47
<i>DLFMT</i>	Format name of data that follows MQDLH	47
<i>DLPAT</i>	Type of application that put message on dead-letter queue	47
<i>DLPAN</i>	Name of application that put message on dead-letter queue	48
<i>DLPD</i>	Date when message was put on dead-letter queue	48
<i>DLPT</i>	Time when message was put on dead-letter queue	48

Overview

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.

MQDLH - Dead-letter header

- 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 *MDCSI* and *MDENC* fields should be set to whatever character set and encoding are used for fields in the MQDLH structure.
- The *MDFMT* field should be set to *FMDLH* to indicate that the data begins with a MQDLH structure.
- The context fields:
 - MDUID*
 - MDACC*
 - MDAID*
 - MDPAT*
 - MDPAN*
 - MDPD*
 - MDPT*
 - MDAOD*

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 *PMDEFEC* 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 *PMPASA* 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 *PMPASI* 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 *PMSETA* option, to preserve the original context information.

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

- The *DLCSI*, *DLENC* and *DLFMT* 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 *DLPAT*, *DLPAN*, *DLPD*, and *DLPT* 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 *MDCSI* field of the message descriptor. Numeric data in the MQDLH structure should be in the data encoding defined by the *MDENC* 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 *MDFMT* field in the message descriptor MQMD; if the field has the value FMDLH, 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

DLSID (4-byte character string)
Structure identifier.

The value must be:

DLSIDV

Identifier for dead-letter header structure.

The initial value of this field is DLSIDV.

DLVER (10-digit signed integer)
Structure version number.

The value must be:

DLVER1

Version number for dead-letter header structure.

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

DLVERC

Current version of dead-letter header structure.

The initial value of this field is DLVER1.

DLREA (10-digit signed integer)
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 FB* or RC* values (for example, RC2053). See the description of the *MDFB* field in “Chapter 10. MQMD - Message descriptor” on page 83 for details of the common FB* values that can occur.

If the value is in the range FBIFST through FBILST, the actual IMS error code can be determined by subtracting FBIERR from the value of the *DLREA* field.

Some FB* values occur only in this field. They relate to repository messages, trigger messages, or transmission-queue messages that have been transferred to the dead-letter queue. These are:

FBABEG

Application cannot be started.

MQDLH - Dead-letter header

An application processing a trigger message was unable to start the application named in the *TMAI* field of the trigger message (see “Chapter 18. MQTM - Trigger message” on page 179).

FBATYP

Application type error.

An application processing a trigger message was unable to start the application because the *TMAI* field of the trigger message is not valid (see “Chapter 18. MQTM - Trigger message” on page 179).

FBNARM

Message is not a repository message.

FBSBCX

Message stopped by channel auto-definition exit.

FBSBMX

Message stopped by channel message exit.

FBTM MQTM structure not valid or missing.

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

FBXQME

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 RCNONE.

DLDQ (48-byte character string)

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 LNQN. The initial value of this field is 48 blank characters.

DLDM (48-byte character string)

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 LNQM. The initial value of this field is 48 blank characters.

DLENC (10-digit signed integer)

Numeric encoding of data that follows MQDLH.

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.

MQDLH - Dead-letter header

When an MQDLH structure is prefixed to the message data, the original data encoding should be preserved by copying it from the *MDENC* field in the message descriptor MQMD to the *DLENC* field in the MQDLH structure. The *MDENC* field in the message descriptor should then be set to the value appropriate to the numeric data in the MQDLH structure.

The value ENNAT can be used for the *DLENC* field in both the MQDLH and MQMD structures.

The initial value of this field is 0.

DLCSI (10-digit signed integer)

Character set identifier of data that follows MQDLH.

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 *MDCSI* field in the message descriptor MQMD to the *DLCSI* field in the MQDLH structure. The *MDCSI* field in the message descriptor should then be set to the value appropriate to the character data in the MQDLH structure.

The value CSQM can be used for the *MDCSI* field in the MQMD structure, but should not be used for the *DLCSI* field in the MQDLH structure, as the queue manager does not replace the value CSQM in the latter field by the value that applies to the queue manager.

The initial value of this field is 0.

DLFMT (8-byte character string)

Format name of data that follows MQDLH.

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 *MDFMT* field in the message descriptor MQMD to the *DLFMT* field in the MQDLH structure. The *MDFMT* field in the message descriptor should then be set to the value FMDLH.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

DLPAT (10-digit signed integer)

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

This field has the same meaning as the *MDPAT* field in the message descriptor MQMD (see “Chapter 10. MQMD - Message descriptor” on page 83 for details).

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

The initial value of this field is 0.

MQDLH - Dead-letter header

DLPAN (28-byte character string)

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

The format of the name depends on the *DLPAT* field. See, also, the description of the *MDPAN* field in “Chapter 10. MQMD - Message descriptor” on page 83.

If it is the queue manager that redirects the message to the dead-letter queue, *DLPAN* contains the first 28 characters of the queue-manager name, padded with blanks if necessary.

The length of this field is given by *LNAPAN*. The initial value of this field is 28 blank characters.

DLPD (8-byte character string)

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 *DLPD* and *DLPT* fields, subject to the system clock being set accurately to GMT.

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

DLPT (8-byte character string)

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 *DLPT*. This happens when leap seconds are inserted into the global time standard.

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

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

Initial values and RPG declaration

Table 14. Initial values of fields in MQDLH

Field name	Name of constant	Value of constant
<i>DLSID</i>	DLSIDV	'DLHb' (See note 1)
<i>DLVER</i>	DLVER1	1
<i>DLREA</i>	RCNONE	0
<i>DLDQ</i>	None	Blanks
<i>DLDM</i>	None	Blanks
<i>DLENC</i>	None	0
<i>DLCSI</i>	None	0
<i>DLFMT</i>	FMNONE	'bbbbbbbb'
<i>DLPAT</i>	None	0
<i>DLPAN</i>	None	Blanks
<i>DLPD</i>	None	Blanks
<i>DLPT</i>	None	Blanks
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQDLH Structure
D*
D* Structure identifier
D DLSID          1      4
D* Structure version number
D DLVER          5      8I 0
D* Reason message arrived on dead-letter (undelivered-message)
D* queue
D DLREA          9      12I 0
D* Name of original destination queue
D DLDQ           13     60
D* Name of original destination queue manager
D DLDM           61     108
D* Numeric encoding of data that follows MQDLH
D DLENC          109    112I 0
D* Character set identifier of data that follows MQDLH
D DLCSI          113    116I 0
D* Format name of data that follows MQDLH
D DLFMT          117    124
D* Type of application that put message on dead-letter
D* (undelivered-message) queue
D DLPAT          125    128I 0
D* Name of application that put message on dead-letter
D* (undelivered-message) queue
D DLPAN          129    156
D* Date when message was put on dead-letter (undelivered-message)
D* queue
D DLPD           157    164
D* Time when message was put on the dead-letter
D* (undelivered-message) queue
D DLPT           165    172

```

MQDLH - Dead-letter header

Chapter 8. MQGMO - Get-message options

The following table summarizes the fields in the structure.

Table 15. Fields in MQGMO

Field	Description	Page
<i>GMSID</i>	Structure identifier	51
<i>GMVER</i>	Structure version number	52
<i>GMOPT</i>	Options that control the action of MQGET	52
<i>GMWI</i>	Wait interval	69
<i>GMSG1</i>	Signal	70
<i>GMSG2</i>	Signal identifier	70
<i>GMRQN</i>	Resolved name of destination queue	70
Note: The remaining fields are not present if <i>GMVER</i> is less than GMVER2.		
<i>GMMO</i>	Options controlling selection criteria used for MQGET	70
<i>GMGST</i>	Flag indicating whether message retrieved is in a group	72
<i>GMSST</i>	Flag indicating whether message retrieved is a segment of a logical message	73
<i>GMSEG</i>	Flag indicating whether further segmentation is allowed for the message retrieved	73
Note: The remaining fields are not present if <i>GMVER</i> is less than GMVER3.		
<i>GMTOK</i>	Message token	73
<i>GMRL</i>	Length of message data returned (bytes)	73

Overview

The current version of MQGMO is given by GMVERC. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow. The declaration of MQGMO provided in the COPY file contains the additional fields, but the initial value provided for the *GMVER* field is GMVER1. To use the additional fields, the application must set the version number to GMVERC. Applications which are intended to be portable between several environments should use a more-recent version MQGMO only if all of those environments support that version.

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

Fields

GMSID (4-byte character string)
Structure identifier.

The value must be:

GMSIDV

Identifier for get-message options structure.

MQGMO - Get-message options

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

GMVER (10-digit signed integer)
Structure version number.

The value must be one of the following:

GMVER1

Version-1 get-message options structure.

GMVER2

Version-2 get-message options structure.

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

GMVER3

Version-3 get-message options structure.

Fields that exist only in the version-3 structure are identified as such in the descriptions that follow.

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

GMVERC

Current version of get-message options structure.

This is always an input field. The initial value of this field is GMVER1.

GMOPT (10-digit signed integer)
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). Combinations of options that are not valid are noted; all other combinations are valid.

GMWT

Wait for message to arrive.

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

If MQGET requests are inhibited, or MQGET requests become inhibited while waiting, the wait is canceled and the call completes with CCFAIL and reason code RC2016, regardless of whether there are suitable messages on the queue.

This option can be used with the GMBRWF or GMBRWN options.

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* GMLK; an MQGET call specifying the GMLK 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,

MQGMO - Get-message options

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 *MDMID* or *MDCID* (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 *MDMID* and *MDCID*.
- 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.

GMWT is ignored if specified with GMBRWC or GMMUC; no error is raised.

GMNWT

Return immediately if no suitable message.

The application is not to wait if no suitable message is available. This is the opposite of the GMWT option, and is defined to aid program documentation. It is the default if neither is specified.

GMSYP

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 GMNSYP is specified, the get request is not within a unit of work.

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

GMBRWF
GMBRWC
GMBRWN
GMLK
GMNSYP
GMPSYP
GMUNLK

MQGMO - Get-message options

GMPSYP

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 PEPER in the *MDPER* field in MQMD.

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

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

GMBRWF
GMBRWC
GMBRWN
GMCMPM
GMNSYP
GMSYP
GMUNLK

GMNSYP

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 the unit of work.

This option is assumed if GMBRWF or GMBRWN is specified.

If neither this option nor GMSYP is specified, the get request is not within a unit of work.

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

GMSYP
GMPSYP

GMBRWF

Browse from start of queue.

When a queue is opened with the OOBROW option, a browse cursor is established, positioned logically before the first message on the queue. Subsequent MQGET calls specifying the GMBRWF, GMBRWN or GMBRWC 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 GMBRWN will search for a suitable message.

An MQGET call with GMBRWF 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 GMBRWN is issued, the browse cursor remains at the position in the queue that the message occupied, even though that position is now empty.

MQGMO - Get-message options

The GMMUC 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 *HOB*J handle. Nor is it moved by a browse MQGET call that returns a completion code of CCFAIL, or a reason code of RC2080.

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

GMBRWF can be specified with any valid combination of the GM* and MO* options that control the processing of messages in groups and segments of logical messages.

If GMLOGO is specified, the messages are browsed in logical order. If that option is omitted, the messages are browsed in physical order. When GMBRWF is specified, it is possible to switch between logical order and physical order, but subsequent MQGET calls using GMBRWN must browse the queue in the same order as the most-recent call that specified GMBRWF 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 GMBRWF 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 CCOK or CCWARN), 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:

- GMBRWC
- GMBRWN
- GMMUC
- GMSYP
- GMPSYP
- GMUNLK

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

GMBRWN

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 GMBRWF or GMBRWN option.

If the message is removed from the queue before the next MQGET call with GMBRWN 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 (MSPRIO), or
- FIFO *regardless* of priority (MSFIFO)

MQGMO - Get-message options

The *MsgDeliverySequence* queue attribute indicates which method applies (see “Chapter 38. Attributes for local queues and model queues” on page 299 for details).

If the queue has a *MsgDeliverySequence* of MSPRIO, 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 GMBRWN. It can only be found after the browse cursor has been reset with GMBRWF (or by reopening the queue).

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

GMBRWN can be specified with any valid combination of the GM* and MO* options that control the processing of messages in groups and segments of logical messages.

If GMLOGO is specified, the messages are browsed in logical order. If that option is omitted, the messages are browsed in physical order. When GMBRWF is specified, it is possible to switch between logical order and physical order, but subsequent MQGET calls using GMBRWN must browse the queue in the same order as the most-recent call that specified GMBRWF for the queue handle. The call fails with reason code RC2259 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 GMLOGO 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 GMBRWN to browse beyond the end of the group, specifying MOSEQN with *MDSEQ* 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 MO* 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 GMBRWN to browse beyond the end of a group.

MQGMO - Get-message options

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:

GMBRWF
GMBRWC
GMMUC
GMSYP
GMPSYP
GMUNLK

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

GMBRWC

Browse message under browse cursor.

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

The message pointed to by the browse cursor is the one that was last retrieved using either the GMBRWF or the GMBRWN 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 GMMUC 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 CCFAIL, or a reason code of RC2080.

If GMBRWC is specified *with* GMLK:

- 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 GMBRWC is specified *without* GMLK:

- 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 GMCMPM is specified with GMBRWC, the browse cursor must identify a message whose *MDOFF* field in MQMD is zero. If this condition is not satisfied, the call fails with reason code RC2246.

MQGMO - Get-message options

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:

GMBRWF
GMBRWN
GMMUC
GMSYP
GMPSYP
GMUNLK

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

GMMUC

Get message under browse cursor.

This option causes the message pointed to by the browse cursor to be retrieved, regardless of the MO* options specified in the *GMMO* 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 GMBRWF or the GMBRWN option.

If GMCMPM is specified with GMMUC, the browse cursor must identify a message whose *MDOFF* field in MQMD is zero. If this condition is not satisfied, the call fails with reason code RC2246.

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

GMBRWF
GMBRWC
GMBRWN
GMUNLK

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.

GMLK

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:

GMBRWF
GMBRWN
GMBRWC

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

- If GMCMPM 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 GMCMPM 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 GMCMPM to retrieve or browse the logical message.

MQGMO - Get-message options

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 GMMUC 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 CCFAIL is returned (or CCWARN with RC2080), 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 GMBRWF or GMBRWN specified (with or without GMLK); the message is unlocked if the call completes with CCOK or CCWARN, but remains locked if the call completes with CCFAIL. However, the following exceptions apply:
 - The message *is not* unlocked if CCWARN is returned with RC2080.
 - The message *is* unlocked if CCFAIL is returned with RC2033.

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

If GMWT 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 GMBRWC (without GMLK); the message is unlocked if the call completes with CCOK or CCWARN, but remains locked if the call completes with CCFAIL. However, the following exception applies:
 - The message *is not* unlocked if CCWARN is returned with RC2080.
- Issuing another MQGET call for this handle with GMUNLK.
- 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 OOBROW, which is needed in order to specify the accompanying browse option.

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

GMSYP
GMPSYP
GMUNLK

GMUNLK

Unlock message.

The message to be unlocked must have been previously locked by an MQGET call with the GMLK option. If there is no message locked for this handle, the call completes with CCWARN and RC2209.

The *MSGDSC*, *BUFLN*, *BUFFER*, and *DATLEN* parameters are not checked or altered if GMUNLK is specified. No message is returned in *BUFFER*.

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

MQGMO - Get-message options

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

GMNWT
GMNSYP

Both of these options are assumed whether specified or not.

GMATM

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 RC2079 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 RC2080 is returned if no other error occurs.

GMFIQ

Fail if queue manager is quiescing.

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

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

- The wait is canceled and the call returns completion code CCFAIL with reason code RC2161.

If GMFIQ is not specified and the queue manager enters the quiescing state, the wait is not canceled.

GMCONV

Convert message data.

This option requests that the application data in the message should be converted, to conform to the *MDCSI* and *MDENC* values specified in the *MSGDSC* parameter on the MQGET call, before the data is copied to the *BUFFER* parameter.

The *MDFMT* 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 F. Data conversion” on page 461 for details of the data-conversion exit.

- If conversion is performed successfully, the *MDCSI* and *MDENC* fields specified in the *MSGDSC* 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

MQGMO - Get-message options

unconverted, and the *MDCSI* and *MDENC* fields in *MSGDSC* are set to the values for the unconverted message. The completion code is *CCWARN* 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 *MDFMT* field described in “Chapter 10. MQMD - Message descriptor” on page 83 for a list of format names for which the queue manager performs the conversion.

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 (*MDMID* 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 (*MDGID* field in *MQMD*), and the same message sequence number (*MDSEQ* field in *MQMD*). The segments are distinguished by differing values for the segment offset (*MDOFF* 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 (*GINONE*), 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 - Get-message options

GMLOGO

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.

If GMLOGO 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 GMLOGO 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 GMLOGO 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 *MDGID*, *MDSEQ*, and *MDOFF* fields in MQMD. However, the application does need to set the *GMSYP* or *GMNSYP* 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 MFMIG flag is returned by the MQGET call. With GMLOGO specified on successive calls, that group remains the current group until a message is returned that has:

- MFLMIG without MFSEG (that is, the last logical message in the group is not segmented), or
- MFLMIG with MFLSEG (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 MFSEG flag is returned by the MQGET call, and that logical message is terminated when the message that has the MFLSEG 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,

MQGMO - Get-message options

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 *GMMO* field:

MOMSGI
MOCORI
MOGRPI

However, these options are effective only when there is no current message group or logical message; see the *GMMO* field described in “Chapter 8. MQGMO - Get-message options” on page 51 for further details.

Table 16 shows the values of the *MDMID*, *MDCID*, *MDGID*, *MDSEQ*, and *MDOFF* 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 columns in the table have the following meanings:

LOG ORD

A “✓” means that the row applies only when the GMLOGO option is specified.

Cur grp

A “✓” means that the row applies only when a current message group exists prior to the call.

A “(✓)” means that the row applies whether or not a current message group exists prior to the call.

Cur log msg

A “✓” means that the row applies only when a current logical message exists prior to the call.

A “(✓)” means that the row applies whether or not a current logical message exists prior to the call.

Other columns

These show the values that the queue manager looks for. “Previous” denotes the value returned for the field in the previous message for the queue handle.

Table 16. 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				
	Cur grp	Cur log msg	<i>MDMID</i>	<i>MDCID</i>	<i>MDGID</i>	<i>MDSEQ</i>	<i>MDOFF</i>
✓			Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>	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 <i>GMMO</i>	Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>	Controlled by <i>GMMO</i>

MQGMO - Get-message options

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 GMLOGO 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 RC2245.

When GMLOGO is specified, the MQGMO supplied on the MQGET call must not be less than GMVER2, and the MQMD must not be less than MDVER2. If this condition is not satisfied, the call fails with reason code RC2256 or RC2257, as appropriate.

If GMLOGO 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 MO* options specified on those calls (see the *GMMO* field described in “Chapter 8. MQGMO - Get-message options” on page 51 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 *MDGID*, *MDSEQ*, *MDOFF*, and *GMMO* fields to the appropriate values, and then issue the MQGET call with *GMSYP* or *GMNSYP* set as desired, but *without* specifying GMLOGO. If this call is successful, the queue manager retains the group and segment information, and subsequent MQGET calls using that queue handle can specify GMLOGO 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.

MQGMO - Get-message options

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

- Each successful MQGET call that does *not* specify GMLOGO 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 GMLOGO 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 CCWARN completion code. Table 17 shows the various cases that can arise. In these cases, if the completion code is not CCOK, the reason code is one of the following (as appropriate):
 RC2241
 RC2242
 RC2245

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 CCOK (assuming no other errors).

Table 17. Outcome when MQGET or MQCLOSE call not consistent with group and segment information

Current call	Previous call	
	MQGET with GMLOGO	MQGET without GMLOGO
MQGET with GMLOGO	CCFAIL	CCFAIL
MQGET without GMLOGO	CCWARN	CCOK
MQCLOSE with an unterminated group or logical message	CCWARN	CCOK

Applications that simply want to retrieve messages and segments in logical order are recommended to specify GMLOGO, 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 GMLOGO option, and this can be achieved by not specifying that option. If this is done, the application must ensure that the *MDMID*, *MDCID*, *MDGID*, *MDSEQ*, and *MDOFF* fields in MQMD, and the MO* options in *GMMO* 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 GMLOGO. 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 GMLOGO, nor the corresponding PMLOGO 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 - Get-message options

GMLOGO can be specified with any of the other GM* options, and with various of the MO* options in appropriate circumstances (see above).

GMCMPM

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 GMCMPM.

To use this option, the application must provide a buffer which is big enough to accommodate the complete message, or specify the GMATM 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 GMCMPM 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 RC2255.

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

MQGMO - Get-message options

Each physical message that 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 *MDMID*, *MDOFF*, and *MDMFL* fields that differ between segments in the logical message. However, if a segment is placed on a dead-letter queue at an intermediate queue manager, the DLQ handler retrieves the message specifying the *GMCONV* option, and this may result in the character set or encoding of the segment being changed. If the DLQ handler successfully sends the segment on its way, the segment may have a character set or encoding that differs from the other segments in the logical message when the segment finally arrives at the destination queue manager.

A logical message consisting of segments in which the *MDCSI* and/or *MDENC* 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 *CCWARN* and reason code *RC2243* or *RC2244*, as appropriate. This happens regardless of whether *GMCONV* is specified. To retrieve the remaining segments, the application must reissue the *MQGET* call without the *GMCMPM* option, retrieving the segments one by one. *GMLOGO* 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 *GMCMPM* 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 *MDMFL* field, which the queue manager sets to indicate that the reassembled message is the only segment.

If *GMCMPM* 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 *GMCMPM*. 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 *RO*D* or *RO*F* 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 *GMCMPM* is specified with *GMMUC* or *GMBRWC*, the browse cursor must be positioned on a message whose *MDOFF* field in *MQMD* has a value of 0. If this condition is not satisfied, the call fails with reason code *RC2246*.

GMCMPM implies *GMASGA*, which need not therefore be specified.

MQGMO - Get-message options

GMCMPPM can be specified with any of the other GM* options apart from GMPSYP, and with any of the MO* options apart from MOOFFS.

GMAMSA

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 GMAMSA 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 RC2033 is returned after the specified wait interval (if any) has expired.

The processing of GMAMSA depends on whether GMLOGO is also specified:

- If both options are specified, GMAMSA has an effect *only* when there is no current group or logical message. If there *is* a current group or logical message, GMAMSA is ignored. This means that GMAMSA can remain on when processing messages in logical order.
- If GMAMSA is specified without GMLOGO, GMAMSA *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.

GMAMSA implies GMASGA, which need not therefore be specified.

GMAMSA can be specified with any of the other GM* options, and with any of the MO* options.

GMASGA

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 GMASGA 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 RC2033 is returned after the specified wait interval (if any) has expired.

The processing of GMASGA depends on whether GMLOGO is also specified:

- If both options are specified, GMASGA has an effect *only* when there is no current logical message. If there *is* a current logical message, GMASGA is ignored. This means that GMASGA can remain on when processing messages in logical order.

MQGMO - Get-message options

- If GMASGA is specified without GMLOGO, GMASGA *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 GMCMPM and GMASGA 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 GMASGA 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 GMASGA 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 GMASGA does not imply that GMCMPM 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.

GMASGA can be specified with any of the other GM* options, and with any of the MO* options.

GMNONE

No options specified.

This value can be used to indicate that no other options have been specified; all options assume their default values. GMNONE 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 *GMOPT* field is GMNWT.

GMWI (10-digit signed integer)

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 *MSGDSC* parameter of the MQGET call; see the *MDMID* field described in “Chapter 10. MQMD - Message descriptor” on page 83 for more details). If no suitable message has arrived after this time has elapsed, the call completes with CCFAIL and reason code RC2033.

GMWI is used in conjunction with the GMWT option. It is ignored if this option is not specified. If it is specified, *GMWI* must be greater than or equal to zero, or the following special value:

WIULIM

Unlimited wait interval.

The initial value of this field is 0.

MQGMO - Get-message options

GMSG1 (10-digit signed integer)
Signal.

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

GMSG2 (10-digit signed integer)
Signal identifier.

This is a reserved field; its value is not significant.

GMRQN (48-byte character string)
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 *LNQN*. The initial value of this field is 48 blank characters.

The remaining fields are not present if *GMVER* is less than *GMVER2*.

GMMO (10-digit signed integer)
Options controlling selection criteria used for MQGET.

These options allow the application to choose which fields in the *MSGDSC* 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 *MSGDSC* 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 *MSGDSC* 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), *GMMO* should be set to *MONONE*.

If *GMLOGO* 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 *MDSEQ* equal to 1 and *MDOFF* 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:
 - MOMSGI*
 - MOCORI*
 - MOGRPI*
- 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 *MO** 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 *MSGDSC* parameter must

MQGMO - Get-message options

match the value of the corresponding field in the message to be returned; the call fails with reason code RC2247 if this condition is not satisfied.

GMMO is ignored if either *GMMUC* or *GMBRWC* is specified.

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

MOMSGI

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 *MDMID* field in the *MSGDSC* 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 *MDMID* field in the *MSGDSC* parameter is ignored, and any message identifier will match.

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

MOCORI

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 *MDCID* field in the *MSGDSC* 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 *MDCID* field in the *MSGDSC* parameter is ignored, and any correlation identifier will match.

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

MOGRPI

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 *MDGID* field in the *MSGDSC* 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 *MDGID* field in the *MSGDSC* parameter is ignored, and any group identifier will match.

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

MOSEQN

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 *MDSEQ* field in

MQGMO - Get-message options

the *MSGDSC* 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 *MDSEQ* field in the *MSGDSC* parameter is ignored, and any message sequence number will match.

MOOFFS

Retrieve message with specified offset.

This option specifies that the message to be retrieved must have an offset that matches the value of the *MDOFF* field in the *MSGDSC* 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 *MDOFF* field in the *MSGDSC* parameter is ignored, and any offset will match.

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

MONONE

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 GMAMSA, GMASGA, and GMCMPM options).

MONONE 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 MOMSGI with MOCORI. This field is not present if *GMVER* is less than GMVER2.

Note: The initial value of the *GMMO* 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 *MDMID* and *MDCID* fields to MINONE and CINONE prior to each MQGET call. The need to reset *MDMID* and *MDCID* can be avoided by setting *GMVER* to GMVER2, and *GMMO* to MONONE.

GMGST (1-byte character string)

Flag indicating whether message retrieved is in a group.

It has one of the following values:

GSNIG

Message is not in a group.

GSMIG

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

GSLMIG

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 GSNIG. This field is not present if *GMVER* is less than GMVER2.

GMSST (1-byte character string)

Flag indicating whether message retrieved is a segment of a logical message.

It has one of the following values:

SSNSEG

Message is not a segment.

SSSEG

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

SSLSEG

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 SSNSEG. This field is not present if *GMVER* is less than GMVER2.

GMSEG (1-byte character string)

Flag indicating whether further segmentation is allowed for the message retrieved.

It has one of the following values:

SEGIHB

Segmentation not allowed.

SEGALW

Segmentation allowed.

This is an output field. The initial value of this field is SEGIHB. This field is not present if *GMVER* is less than GMVER2.

GMRE1 (1-byte character string)

Reserved.

This is a reserved field. The initial value of this field is a blank character. This field is not present if *GMVER* is less than GMVER2.

The remaining fields are not present if *GMVER* is less than GMVER3.

GMTOK (16-byte bit string)

Message token.

This is a reserved field; its value is not significant. The following special value is defined:

MTKNON

No message token.

The value is binary zero for the length of the field.

The length of this field is given by LNMTOK. The initial value of this field is MTKNON. This field is not present if *GMVER* is less than GMVER3.

GMRL (10-digit signed integer)

Length of message data returned (bytes).

MQGMO - Get-message options

This is a reserved field; its value is not significant. The following special value is defined:

RLUNDF

Length of returned data not defined.

On OS/390, the value returned for the *GMRL* field is always RLUNDF.

The initial value of this field is RLUNDF. This field is not present if *GMVER* is less than GMVER3.

Initial values and RPG declaration

Table 18. Initial values of fields in MQGMO

Field name	Name of constant	Value of constant
<i>GMSID</i>	GMSIDV	'GMOb' (See note 1)
<i>GMVER</i>	GMVER1	1
<i>GMOPT</i>	GMNWT	0
<i>GMWI</i>	None	0
<i>GMSG1</i>	None	0
<i>GMSG2</i>	None	0
<i>GMRQN</i>	None	Blanks
<i>GMMO</i>	MOMSGI + MOCORI	3
<i>GMGST</i>	GSNIG	'b'
<i>GMSST</i>	SSNSEG	'b'
<i>GMSEG</i>	SEGIHB	'b'
<i>GMRE1</i>	None	'b'
<i>GMTOK</i>	MTKNON	Nulls
<i>GMRL</i>	RLUNDF	-1

Notes:
1. The symbol 'b' represents a single blank character.

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQGMO Structure
D*
D* Structure identifier
D GMSID 1 4
D* Structure version number
D GMVER 5 8I 0
D* Options that control the action of MQGET
D GMOPT 9 12I 0
D* Wait interval
D GMWI 13 16I 0
D* Signal
D GMSG1 17 20I 0
D* Signal identifier
D GMSG2 21 24I 0
D* Resolved name of destination queue
D GMRQN 25 72
D* Options controlling selection criteria used for MQGET
D GMMO 73 76I 0

```


RPG declaration

```
D* Flag indicating whether message retrieved is in a group
D  GMGST          77    77
D* Flag indicating whether message retrieved is a segment of a
D* logical message
D  GMSST          78    78
D* Flag indicating whether further segmentation is allowed for the
D* message retrieved
D  GMSEG          79    79
D* Reserved
D  GMRE1          80    80
D* Message token
D  GMTOK          81    96
D* Length of message data returned (bytes)
D  GMRL           97    100I 0
```

RPG declaration

Chapter 9. MQIIH - IMS bridge header

The following table summarizes the fields in the structure.

Table 19. Fields in MQIIH

Field	Description	Page
<i>IISID</i>	Structure identifier	78
<i>IIVER</i>	Structure version number	78
<i>IILEN</i>	Length of MQIIH structure	78
<i>IIFMT</i>	MQ format name of data that follows MQIIH	78
<i>IIFLG</i>	Flags	78
<i>IILTO</i>	Logical terminal override	79
<i>IIMMN</i>	Message format services map name	79
<i>IIRFM</i>	MQ format name of reply message	79
<i>IIAUT</i>	RACF™ password or passticket	79
<i>IITID</i>	Transaction instance identifier	79
<i>IITST</i>	Transaction state	80
<i>IICMT</i>	Commit mode	80
<i>IISEC</i>	Security scope	80

Overview

The MQIIH structure describes the information that must be present at the start of a message sent to the IMS bridge through MQSeries for OS/390. The format name of this structure is FMIMS.

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 *IICSI* and *IIENC* fields in the MQIIH structure cannot be used to specify the character set and encoding of the application message data.

Fields

IISID (4-byte character string)
Structure identifier.

The value must be:

IISIDV

Identifier for IMS information header structure.

The initial value of this field is IISIDV.

IIVER (10-digit signed integer)
Structure version number.

The value must be:

IIVER1

Version number for IMS information header structure.

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

IIVERC

Current version of IMS information header structure.

The initial value of this field is IIVER1.

IILEN (10-digit signed integer)
Length of MQIIH structure.

The value must be:

IILEN1

Length of IMS information header structure.

The initial value of this field is IILEN1.

IIENC (10-digit signed integer)
Reserved.

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

IICSI (10-digit signed integer)
Reserved.

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

IIFMT (8-byte character string)
MQ format name of data that follows MQIIH.

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 *MDFMT* field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

IIFLG (10-digit signed integer)
Flags.

The value must be:

IINONE

No flags.

The initial value of this field is IINONE.

IILTO (8-byte character string)

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 LNLTOV. The initial value of this field is 8 blank characters.

IIMMN (8-byte character string)

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 LNMFMN. The initial value of this field is 8 blank characters.

IIRFM (8-byte character string)

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 *MDFMT* field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

IIAUT (8-byte character string)

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:

IAUNON

No authentication.

The length of this field is given by LNAUTH. The initial value of this field is IAUNON.

IITID (16-byte bit string)

Transaction instance identifier.

This field is used by output messages from IMS so is ignored on first input. If *IITST* is set to ITSIC, this must be provided in the next input, and all

MQIIH - IMS bridge header

subsequent inputs, to enable IMS to correlate the messages to the correct conversation. The following special value may be used:

ITINON

No transaction instance id.

The length of this field is given by LNTIID. The initial value of this field is ITINON.

IITST (1-byte character string)

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:

ITSIC In conversation.

ITSNIC

Not in conversation.

ITSARC

Return transaction state data in architected form.

This value is used only with the IMS /DISPLAY TRAN command. It causes the transaction state data to be returned in the IMS architected form instead of character form. See the *MQSeries Application Programming Guide* for further details.

The initial value of this field is ITSNIC.

IICMT (1-byte character string)

Commit mode.

See the *OTMA Reference* for more information about IMS commit modes. The value must be one of the following:

ICMCTS

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.

ICMSTC

Send then commit.

The initial value of this field is ICMCTS.

IISEC (1-byte character string)

Security scope.

This indicates the desired IMS security processing. The following values are defined:

ISSCHK

Check security scope.

An ACEE is built in the control region, but not in the dependent region.

ISSFUL

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 ISSFUL, you must ensure that the user ID for which the ACEE is built has access to the resources used in the dependent region.

If neither ISSCHK nor ISSFUL is specified for this field, ISSCHK is assumed.

The initial value of this field is ISSCHK.

IIRSV (1-byte character string)

Reserved.

This is a reserved field; it must be blank.

Initial values and RPG declaration

Table 20. Initial values of fields in MQIIH

Field name	Name of constant	Value of constant
<i>IISID</i>	IISIDV	'IIHb' (See note 1)
<i>IIVER</i>	IIVER1	1
<i>IILEN</i>	IILEN1	84
<i>IIENC</i>	None	0
<i>IICSI</i>	None	0
<i>IIFMT</i>	FMNONE	'bbbbbbbb'
<i>IIFLG</i>	IINONE	0
<i>IILTO</i>	None	'bbbbbbbb'
<i>IIMMN</i>	None	'bbbbbbbb'
<i>IIRFM</i>	FMNONE	'bbbbbbbb'
<i>IIAUT</i>	IAUNON	'bbbbbbbb'
<i>IITID</i>	ITINON	Nulls
<i>IITST</i>	ITSNIC	'b'
<i>IICMT</i>	ICMCTS	'0'
<i>IISEC</i>	ISSCHK	'C'
<i>IIRSV</i>	None	'b'
Notes:		
1. The symbol 'b' represents a single blank character.		

MQIIH - IMS bridge header

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQIIH Structure
D*
D* Structure identifier
D IISID          1      4
D* Structure version number
D IIVER          5      8I 0
D* Length of MQIIH structure
D IILEN          9      12I 0
D* Reserved
D IIENC          13     16I 0
D* Reserved
D IICSI          17     20I 0
D* MQ format name of data that follows MQIIH
D IIFMT          21     28
D* Flags
D IIFLG          29     32I 0
D* Logical terminal override
D IILTO          33     40
D* Message format services map name
D IIMMN          41     48
D* MQ format name of reply message
D IIRFM          49     56
D* RACF password or passticket
D IIAUT          57     64
D* Transaction instance identifier
D IITID          65     80
D* Transaction state
D IITST          81     81
D* Commit mode
D IICMT          82     82
D* Security scope
D IISEC          83     83
D* Reserved
D IIRSV          84     84
```


Chapter 10. MQMD - Message descriptor

The following table summarizes the fields in the structure.

Table 21. Fields in MQMD

Field	Description	Page
<i>MDSID</i>	Structure identifier	85
<i>MDVER</i>	Structure version number	85
<i>MDREP</i>	Options for report messages	85
<i>MDMT</i>	Message type	95
<i>MDEXP</i>	Message lifetime	96
<i>MDFB</i>	Feedback or reason code	98
<i>MDENC</i>	Numeric encoding of message data	102
<i>MDCSI</i>	Character set identifier of message data	102
<i>MDFMT</i>	Format name of message data	103
<i>MDPRI</i>	Message priority	107
<i>MDPER</i>	Message persistence	108
<i>MDMID</i>	Message identifier	109
<i>MDCID</i>	Correlation identifier	111
<i>MDBOC</i>	Backout counter	112
<i>MDRQ</i>	Name of reply queue	112
<i>MDRM</i>	Name of reply queue manager	113
<i>MDUID</i>	User identifier	114
<i>MDACC</i>	Accounting token	115
<i>MDAID</i>	Application data relating to identity	116
<i>MDPAT</i>	Type of application that put the message	117
<i>MDPAN</i>	Name of application that put the message	119
<i>MDPD</i>	Date when message was put	119
<i>MDPT</i>	Time when message was put	120
<i>MDAOD</i>	Application data relating to origin	121
Note: The remaining fields are not present if <i>MDVER</i> is less than MDVER2.		
<i>MDGID</i>	Group identifier	121
<i>MDSEQ</i>	Sequence number of logical message within group	122
<i>MDOFF</i>	Offset of data in physical message from start of logical message	123
<i>MDMFL</i>	Message flags	124
<i>MDOLN</i>	Length of original message	128

Overview

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 ENNAT).

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 GMCONV option described in “Chapter 8. MQGMO - Get-message options” on page 51
- The usage note describing GMCONV in “Chapter 30. MQGET - Get message” on page 231
- The *MQSeries Application Programming Guide*

When a message is on a transmission queue, some of the fields in MQMD are set to particular values; see “Chapter 21. MQXQH - Transmission queue header” on page 193 for details.

The current version of MQMD is MDVER2. Fields that exist only in the version-2 structure are identified as such in the descriptions that follow. The declaration of MQMD provided in the COPY file contains the new fields, but the initial value provided for the *MDVER* field is MDVER1; this ensures compatibility with existing applications. To use the new fields, the application must set the version number to MDVER2. 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.

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 *MDFMT* field in MQMD to FMMDE 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 *MDFMT* field in MQMD will have the value FMMDE 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 25 on page 135.

This structure is an input/output parameter for the MQGET, MQPUT, and MQPUT1 calls.

Fields

MDSID (4-byte character string)
Structure identifier.

The value must be:

MDSIDV

Identifier for message descriptor structure.

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

MDVER (10-digit signed integer)
Structure version number.

The value must be one of the following:

MDVER1

Version-1 message descriptor structure.

MDVER2

Version-2 message descriptor structure.

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 4 on page 272 for the MQPUT call.

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

MDVERC

Current version of message descriptor structure.

This is always an input field. The initial value of this field is MDVER1.

MDREP (10-digit signed integer)
Options for report messages.

A report message is a message about another message, used to inform an application about expected or unexpected events that relate to the original message. The *MDREP* 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:

MQMD - Message descriptor

- 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).

The application that receives the report message can determine the reason the report was generated by examining the *MDFB* field in the MQMD; see the *MDFB* field for more details.

Exception options: You can specify one of the options listed below to request an exception report message.

ROEXC

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:
 - RODLQ (this causes the original message to be placed on the dead-letter queue).
 - RODISC (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 RODLQ 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.

MQMD - Message descriptor

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 ROEXC, ROEXCD, and ROEXCF.

ROEXCD

Exception reports with data required.

This is the same as ROEXC, 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 ROEXC, ROEXCD, and ROEXCF.

ROEXCF

Exception reports with full data required.

This is the same as ROEXC, 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 ROEXC, ROEXCD, and ROEXCF.

Expiration options: You can specify one of the options listed below to request an expiration report message.

ROEXP

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 *MDEXP* 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 ROEXC* options is specified).

Message data from the original message is not included with the report message.

Do not specify more than one of ROEXP, ROEXPD, and ROEXPF.

ROEXPD

Expiration reports with data required.

This is the same as ROEXP, 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 ROEXP, ROEXPD, and ROEXPF.

ROEXPF

Expiration reports with full data required.

This is the same as ROEXP, 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 ROEXP, ROEXPD, and ROEXPF.

MQMD - Message descriptor

Confirm-on-arrival options: You can specify one of the options listed below to request a confirm-on-arrival report message.

ROCOA

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 *MDFMT* field in the message descriptor is FMXQH or FMDLH. 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 ROCOA, ROCOAD, and ROCOAF.

ROCOAD

Confirm-on-arrival reports with data required.

This is the same as ROCOA, 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 ROCOA, ROCOAD, and ROCOAF.

ROCOAF

Confirm-on-arrival reports with full data required.

This is the same as ROCOA, 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 ROCOA, ROCOAD, and ROCOAF.

Confirm-on-delivery options: You can specify one of the options listed below to request a confirm-on-delivery report message.

ROCOD

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 generated if the *MDFMT* field in the message descriptor is FMDLH. This prevents a COD report being generated if the message is undeliverable and put on a dead-letter queue.

ROCOD is not valid if the destination queue is an XCF queue.

Do not specify more than one of ROCOD, ROCODD, and ROCODF.

ROCODD

Confirm-on-delivery reports with data required.

This is the same as ROCOD, 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.

If GMATM is specified on the MQGET call for the original message, and the message returned is truncated, the amount of message data placed in the report message depends on the environment:

- On OS/390, it is the minimum of:
 - The length of the original message
 - The length of the buffer used to retrieve the message
 - 100 bytes.
- In other environments, it is the minimum of:
 - The length of the original message
 - 100 bytes.

ROCODD is not valid if the destination queue is an XCF queue.

Do not specify more than one of ROCOD, ROCODD, and ROCODF.

ROCODF

Confirm-on-delivery reports with full data required.

This is the same as ROCOD, except that all of the application message data from the original message is included in the report message.

ROCODF is not valid if the destination queue is an XCF queue.

Do not specify more than one of ROCOD, ROCODD, and ROCODF.

Action-notification options: You can specify one or both of the options listed below to request that the receiving application send a positive-action or negative-action report message.

ROPAN

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.

RONAN

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

MQMD - Message descriptor

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 options listed below to control how the *MDMID* of the report message (or of the reply message) is to be set.

RONMI

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 *MDMID* is to be generated for the report or reply message.

ROPMI

Pass message identifier.

If a report or reply is generated as a result of this message, the *MDMID* of this message is to be copied to the *MDMID* of the report or reply message.

If this option is not specified, RONMI is assumed.

Correlation-identifier options: You can specify one of the options listed below to control how the *MDCID* of the report message (or of the reply message) is to be set.

ROCMTC

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 *MDMID* of this message is to be copied to the *MDCID* of the report or reply message.

ROPCI

Pass correlation identifier.

If a report or reply is generated as a result of this message, the *MDCID* of this message is to be copied to the *MDCID* of the report or reply message.

If this option is not specified, ROCMTC is assumed.

Servers replying to requests or generating report messages are recommended to check whether the ROPMI or ROPCI 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 options listed below to control the disposition of the original message when it cannot be delivered to the destination queue. These options apply only to those situations that would result in an exception report message being generated if one had been requested by the sending application. The application can set the disposition options independently of requesting exception reports.

RODLQ

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.

RODISC

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.

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 RODISC with ROEXCF.

Default option: You can specify the following if no report options are required:

RONONE

No reports required.

This value can be used to indicate that no other options have been specified. RONONE 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, 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 *MDREP* options are set, no report messages are generated by the queue manager or message channel agent (MCA).

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 E. Report options” on page 457 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 *MDRQ* field. When a report message is received, the nature of the report can be determined by examining the *MDFB* field in the message descriptor.

If the queue manager or MCA 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:

MQMD - Message descriptor

- 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 *MDRQ* queue is opened and the report message put using the authority of the *MDUID* in the MQMD of the message causing the report, except in the following cases:

- Exception reports generated by a receiving MCA are put with whatever authority the MCA used when it tried to put the message causing the report. The *CDPA* channel attribute determines the user identifier used.
- COA reports generated by the queue manager are put with whatever authority was used when the message causing the report was put on the queue manager generating the report. For example, if the message was put by a receiving MCA using the MCA's user identifier, the queue manager puts the COA report using the MCA's user identifier.

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 identifier 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 *MDFMT* of the original message is *FMXQH*, 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 an MCA 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 MFSEGA 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 *MDGID* field in the report message can be used to correlate the multiple reports with the group identifier of the original message, and the *MDFB* field used to identify the type of each report message.

If GMLOGO 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 GMCMPM option (optionally with GMATM). GMCMPM 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 GMCMPM 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 ROCOD* options, but at least one segment does not, it will not be possible

MQMD - Message descriptor

to use the GMCMPM option to retrieve the report messages with a single MQGET call, or use the GMASGA 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 RO*D or RO*F options. However, be aware that if RO*D 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 (MCA) 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
<i>MDSID</i>	MDSIDV
<i>MDVER</i>	MDVER2
<i>MDREP</i>	RONONE
<i>MDMT</i>	MTRPRT
<i>MDEXP</i>	EIULIM
<i>MDFB</i>	As appropriate for the nature of the report (FBCOA, FBCOD, FBEXP, or an RC* value)
<i>MDENC</i>	Copied from the original message descriptor
<i>MDCSI</i>	Copied from the original message descriptor
<i>MDFMT</i>	Copied from the original message descriptor
<i>MDPRI</i>	Copied from the original message descriptor
<i>MDPER</i>	Copied from the original message descriptor
<i>MDMID</i>	As specified by the report options in the original message descriptor
<i>MDCID</i>	As specified by the report options in the original message descriptor
<i>MDBOC</i>	0
<i>MDRQ</i>	Blanks
<i>MDRM</i>	Name of queue manager
<i>MDUID</i>	As set by the PMPASI option
<i>MDACC</i>	As set by the PMPASI option
<i>MDAID</i>	As set by the PMPASI option
<i>MDPAT</i>	ATQM, or as appropriate for the message channel agent
<i>MDPAN</i>	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.
<i>MDPD</i>	Date when report message is sent
<i>MDPT</i>	Time when report message is sent
<i>MDAOD</i>	Blanks
<i>MDGID</i>	Copied from the original message descriptor
<i>MDSEQ</i>	Copied from the original message descriptor
<i>MDOFF</i>	Copied from the original message descriptor
<i>MDMFL</i>	Copied from the original message descriptor

MDOLN Copied from the original message descriptor if not *OLUNDF*, 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 *MDRM* 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 *PMPASI*.

Analyzing the report field: The *MDREP* 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 458.

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 *RONONE*.

MDMT (10-digit signed integer)
Message type.

This indicates the type of the message. Message types are grouped as follows:

MTSFST

Lowest value for system-defined message types.

MTSLST

Highest value for system-defined message types.

The following values are currently defined within the system range:

MTDGRM

Message not requiring a reply.

The message is one that does not require a reply.

MTRQST

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 *MDRQ* field. The *MDREP* field indicates how the *MDMID* and *MDCID* of the reply are to be set.

MTRPLY

Reply to an earlier request message.

The message is the reply to an earlier request message (*MTRQST*). The message should be sent to the queue indicated by the *MDRQ* field of the request message. The *MDREP* field of the request should be used to control how the *MDMID* and *MDCID* of the reply are set.

Note: The queue manager does not enforce the request-reply relationship; this is an application responsibility.

MTRPRT

Report message.

The message is reporting on some expected or unexpected occurrence, usually related to some other message (for example, a request message

MQMD - Message descriptor

was received which contained data that was not valid). The message should be sent to the queue indicated by the *MDRQ* field of the message descriptor of the original message. The *MDFB* field should be set to indicate the nature of the report. The *MDREP* field of the original message can be used to control how the *MDMID* and *MDCID* of the report message should be set.

Report messages generated by the queue manager or message channel agent are always sent to the *MDRQ* queue, with the *MDFB* and *MDCID* 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:

MTAFST

Lowest value for application-defined message types.

MTALST

Highest value for application-defined message types.

For the MQPUT and MQPUT1 calls, the *MDMT* value must be within either the system-defined range or the application-defined range; if it is not, the call fails with reason code RC2029.

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 MTDGRM.

MDEXP (10-digit signed integer)
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 *MDEXP* 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. In the current implementations, the message is discarded when a browse or nonbrowse MQGET call occurs that would have returned the message had it not already expired. For example, a nonbrowse MQGET call with the *GMMO* field in MQGMO set to MONONE reading from a FIFO ordered queue will cause all the expired messages to be discarded up to the first unexpired message. With a priority ordered queue, the same call will

discard expired messages of higher priority and messages of an equal priority that arrived on the queue before the first unexpired message.

A message that has expired is never returned to an application (either by a browse or a non-browse MQGET call), so the value in the *MDEXP* field of the message descriptor after a successful MQGET call is either greater than zero, or the special value EIULIM.

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 ROEXP* 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 *MDEXP* time of zero, the MQPUT or MQPUT1 call fails with reason code RC2013; 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 GMLK, the message may become eligible to be discarded before it can be retrieved by an MQGET call with GMMUC; reason code RC2034 is returned on this subsequent MQGET call if that happens.
7. When a request message with an expiry time greater than zero is retrieved, the application can take one of the following actions when it sends the reply message:
 - Copy the remaining expiry time from the request message to the reply message.
 - Set the expiry time in the reply message to an explicit value greater than zero.
 - Set the expiry time in the reply message to EIULIM.

The action to take depends on the design of the application suite. However, the default action for putting messages to a dead-letter

MQMD - Message descriptor

(undelivered-message) queue should be to preserve the remaining expiry time of the message, and to continue to decrement it.

8. Trigger messages are always generated with EIULIM.
9. A message (normally on a transmission queue) which has a *MDFMT* name of FMXQH has a second message descriptor within the MQXQH. It therefore has two *MDEXP* 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 *MDEXP* 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 FMXQH (but the queue manager does not enforce this). In this case the application need not set the values of these two *MDEXP* fields to be the same. (The queue manager does not check that the *MDEXP* 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 *MDFMT* name of FMXQH is retrieved from a queue (whether this is a normal or a transmission queue), the queue manager decrements *both* these *MDEXP* fields with the time spent waiting on the queue. No error is raised if the message data is not long enough to include the *MDEXP* field in the MQXQH.
- The queue manager uses the *MDEXP* 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 *MDEXP* fields were different, it is therefore possible for the *MDEXP* 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 *MDEXP* field in the MQXQH has elapsed. In this case the *MDEXP* field in the MQXQH is set to zero.

The following special value is recognized:

EIULIM

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 EIULIM.

MDFB (10-digit signed integer)

Feedback or reason code.

This is used with a message of type MTRPRT to indicate the nature of the report, and is only meaningful with that type of message. The field can contain one of the FB* values, or one of the RC* values. Feedback codes are grouped as follows:

FBNONE

No feedback provided.

FBSFST

Lowest value for system-generated feedback.

FBSLST

Highest value for system-generated feedback.

The range of system-generated feedback codes FBSFST through FBSLST includes the general feedback codes listed below (FB*), and also the reason codes (RC*) that can occur when the message cannot be put on the destination queue.

FBAFST

Lowest value for application-generated feedback.

FBALST

Highest value for application-generated feedback.

Applications that generate report messages should not use feedback codes in the system range (other than FBQUIT), 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 either be FBNONE, or be within the system range or application range. This is checked whatever the value of *MDMT*.

General feedback codes:**FBCOA**

Confirmation of arrival on the destination queue (see ROCOA).

FBCOD

Confirmation of delivery to the receiving application (see ROCOD).

FBEXP

Message expired.

Message was discarded because it had not been removed from the destination queue before its expiry time had elapsed.

FBPAN

Positive action notification (see ROPAN).

FBNAN

Negative action notification (see RONAN).

FBQUIT

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 MTRPRT 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.

IMS-bridge feedback codes: When the IMS bridge receives a nonzero IMS-OTMA sense code, the IMS bridge converts the sense code from hexadecimal to decimal, adds the value FBIERR (300), and places the result in the *MDFB* field of the reply message. This results in the feedback code having a value in the range FBIFST (301) through FBILST (399) when an IMS-OTMA error has occurred.

MQMD - Message descriptor

The following feedback codes can be generated by the IMS bridge:

FBDLZ

Data length zero.

A segment length was zero in the application data of the message.

FBDLN

Data length negative.

A segment length was negative in the application data of the message.

FBDLTB

Data length too big.

A segment length was too big in the application data of the message.

FBBUFO

Buffer overflow.

The value of one of the length fields would cause the data to overflow the MQSeries message buffer.

FBLOB1

Length in error by one.

The value of one of the length fields was one byte too short.

FBIIH MQIIH structure not valid or missing.

The *MDFMT* field in MQMD specifies FMIMS, but the message does not begin with a valid MQIIH structure.

FBNAFI

Userid not authorized for use in IMS.

The user ID contained in the message descriptor MQMD, or the password contained in the *IIAUT* field in the MQIIH structure, failed the validation performed by the IMS bridge. As a result the message was not passed to IMS.

FBIERR

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.

FBIFST

Lowest value for IMS-generated feedback.

IMS-generated feedback codes occupy the range FBIFST (300) through FBILST (399). The IMS-OTMA sense code itself is *MDFB* minus FBIERR.

FBILST

Highest value for IMS-generated feedback.

CICS-bridge feedback codes: The following feedback codes can be generated by the CICS bridge:

FBCAAB

Application abended.

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

FBCANS

Application cannot be started.

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

FBCBRF

CICS bridge terminated abnormally without completing normal error processing.

FBCCESE

Character set identifier not valid.

FBCIHE

CICS information header structure missing or not valid.

FBCCAE

Length of CICS commarea not valid.

FBCCEI

Correlation identifier not valid.

FBCDLQ

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.

FBCENE

Encoding not valid.

FBCINE

CICS bridge encountered an unexpected error.

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

FBCNTA

User identifier not authorized or password not valid.

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

FBCUBO

Unit of work backed out.

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

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

FBCUWE

Unit-of-work control field *CIUOW* not valid.

MQ reason codes: For exception report messages, *MDFB* contains an MQ reason code. Among possible reason codes are:

RC2051

(2051, X'803') Put calls inhibited for the queue.

RC2053

(2053, X'805') Queue already contains maximum number of messages.

MQMD - Message descriptor

RC2035

(2035, X'7F3') Not authorized for access.

RC2056

(2056, X'808') No space available on disk for queue.

RC2048

(2048, X'800') Message on a temporary dynamic queue cannot be persistent.

RC2031

(2031, X'7EF') Message length greater than maximum for queue manager.

RC2030

(2030, X'7EE') Message length greater than maximum for queue.

For a full list of reason codes, see “Reason codes” on page 357.

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 FBNONE.

MDENC (10-digit signed integer)

Numeric encoding of message data.

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:

ENNAT

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 ENNAT. Applications that retrieve messages should compare this field against the value ENNAT; if the values differ, the application may need to convert numeric data in the message. See “Appendix D. Machine encodings” on page 453 for details of how this field is constructed.

If the GMCONV 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 ENNAT.

MDCSI (10-digit signed integer)

Character set identifier of message data.

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 "Chapter 43. Attributes for the queue manager" on page 323 for details of this attribute.

The following values are defined:

CSQM

Queue manager's character set identifier.

Character data in the message is in the queue manager's character set.

CSINHT

Inherit character-set identifier of this structure.

Character data in the message is in the same character set as MQMD; this is the queue-manager's character set.

CSEMBD

Embedded character set identifier.

Character data in the message is in a character set whose identifier is contained within the message data itself. There can be any number of character-set identifiers embedded within the message data, applying to different parts of the data. This value must be used for PCF messages that contain data in a mixture of character sets. PCF messages have a format name of FMPCF.

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 CSQM in the *MDCSI* field in the MQMD sent with the message to the value of the queue manager's *CodedCharSetId* attribute; as a result, the value CSQM is never returned by the MQGET call. The *MDCSI* field in the MQMD specified on the MQPUT or MQPUT1 call is not altered. 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 GMCONV 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 CSQM 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 CSQM.

MDFMT (8-byte character string)

Format name of message data.

MQMD - Message descriptor

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:

FMNONE

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 GMCONV is specified on the MQGET call for a message that has a format name of FMNONE, and the character set or encoding of the message differs from that specified in the *MSGDSC* parameter, the message is still returned in the *BUFFER* parameter (assuming no other errors), but the call completes with completion code CCWARN and reason code RC2110.

FMADMN

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 GMCONV option is specified on the MQGET call. Refer to the *MQSeries Programmable System Management* book for more information about using programmable command format messages.

FMCIICS

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 *CIFMT* field in the MQCIH structure.

FMCMND1

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 GMCONV option is specified on the MQGET call.

FMCMD2

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 GMCONV option is specified on the MQGET call.

FMDLH

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 *DLFMT* field in the MQDLH structure; see “Chapter 7. MQDLH - Dead-letter header” on page 43 for details of this structure. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

COA and COD reports are not generated for messages which have a *MDFMT* of FMDLH.

FMDH

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 *DHFMT* field in the MQDH structure; see “Chapter 6. MQDH - Distribution header” on page 37 for details of this structure. Messages with format FMDH can be converted if the GMCONV option is specified on the MQGET call.

FMEVNT

Event message.

The message is an MQ event message that reports an event that occurred. Messages of this format can be converted if the GMCONV option is specified on the MQGET call. Event messages have the same structure as programmable commands; Refer to the *MQSeries Programmable System Management* book for more information about this structure.

FMIMS

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 *IIFMT* field in the MQIIH structure. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

FMIMVS

IMS variable string.

The message is an IMS variable string, which is a string of the form 11zzccc, where:

- 11 is a 2-byte length field specifying the total length of the IMS variable string item. This length is equal to the length of 11 (2 bytes), plus the length of zz (2 bytes), plus the length of the character string itself. 11 is a 2-byte binary integer in the encoding specified by the *MDENC* field.

MQMD - Message descriptor

- zz** is a 2-byte field containing flags that are significant to IMS. *zz* is a byte string consisting of two 1-byte bit string 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 1-4 characters. *ccc* is in the character set specified by the *MDCSI* field.

Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMMDE

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 *MEFMT*, *MECSI*, and *MEENC* fields in the *MQMDE*. See “Chapter 11. *MQMDE* - Message descriptor extension” on page 131 for details of this structure. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMPCF

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 *GMCONV* option is specified on the *MQGET* call. Refer to the *MQSeries Programmable System Management* book for more information about using programmable command format messages.

FMRMH

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 *RMFMT*, *RMCSI*, and *RMENC* fields in the *MQRMH*. See “Chapter 16. *MQRMH* - Message reference header” on page 169 for details of this structure. Messages of this format can be converted if the *GMCONV* option is specified on the *MQGET* call.

FMRFH

Rules and formatting header.

The message data begins with the rules and formatting header *MQRFH*, and is optionally followed by other data. The format name, character set, and encoding of the data (if any) is given by the *RFMT*, *RFCSI*, and *RFENC* fields in the *MQRFH*.

FMSTR

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 *GMCONV* option is specified on the *MQGET* call.

FMTM

Trigger message.

MQMD - Message descriptor

The message is a trigger message, described by the MQTM structure; see “Chapter 18. MQTM - Trigger message” on page 179 for details of this structure. Messages of this format can be converted if the GMCONV option is specified on the MQGET call.

FMWIH

Work information header.

The message data begins with the work information header MQWIH, which is followed by the application data. The format name of the application data is given by the *WIFMT* field in the MQWIH structure.

FMXQH

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 *MDFMT* field in the MQMD structure which is part of the transmission queue header MQXQH. See “Chapter 21. MQXQH - Transmission queue header” on page 193 for details of this structure.

COA and COD reports are not generated for messages which have a *MDFMT* of FMXQH.

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 LNFMT. The initial value of this field is FMNONE.

MDPRI (10-digit signed integer)

Message priority.

For the MQPUT and MQPUT1 calls, the value must be greater than or equal to zero; zero is the lowest priority.

The following special value can also be used:

PRQDEF

Default priority for queue.

- If the queue is a cluster queue, the priority for the message is taken from the *DefPriority* attribute as defined at the *destination* queue manager that owns the particular instance of the queue on which the message is placed. Usually, all of the instances of a cluster queue have the same value for the *DefPriority* attribute, although this is not mandated.

The value of *DefPriority* is copied into the *MDPRI* field when the message is placed on the destination queue. If *DefPriority* is changed subsequently, messages that have already been placed on the queue are not affected.

- If the queue is not a cluster queue, the priority for the message is taken from the *DefPriority* attribute as defined at the *local* queue manager, even if the destination queue manager is remote.

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. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue

MQMD - Message descriptor

- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value of *DefPriority* is copied into the *MDPRI* field when the message is put. If *DefPriority* is changed subsequently, messages that have already been put are not affected.

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 CCWARN and reason code RC2049. However, the *MDPRI* field retains the value specified by the application which put the message.

The value returned by the MQGET call is always greater than or equal to zero; the value PRQDEF 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 PRQDEF.

MDPER (10-digit signed integer)
Message persistence.

For the MQPUT and MQPUT1 calls, the value must be one of the following:

PEPER

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 placed on temporary dynamic queues; persistent messages can however be placed on permanent dynamic queues, and predefined queues.

Once a persistent message has been put (or the unit of work committed, if the MQPUT or MQPUT1 call 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 MQGET call 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.

PENPER

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.

PEQDEF

Message has default persistence.

- If the queue is a cluster queue, the persistence of the message is taken from the *DefPersistence* attribute defined at the *destination* queue manager that owns the particular instance of the queue on which the message is placed.

Usually, all of the instances of a cluster queue have the same value for the *DefPersistence* attribute, although this is not mandated.

The value of *DefPersistence* is copied into the *MDPER* field when the message is placed on the destination queue. If *DefPersistence* is changed subsequently, messages that have already been placed on the queue are not affected.

- If the queue is not a cluster queue, the persistence of the message is taken from the *DefPersistence* attribute defined at the *local* queue manager, even if the destination queue manager is remote.

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. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value of *DefPersistence* is copied into the *MDPER* field when the message is put. If *DefPersistence* is changed subsequently, messages that have already been put are not affected.

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 PEPER or PENPER.

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 PEQDEF.

MDMID (24-byte bit string)

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 MINONE or PMNMID is specified by the application, the queue manager generates a unique message identifier ¹

1. A *MDMID* 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

MQMD - Message descriptor

when the message is put, 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 *MDMID* field in MQMD is unchanged on return from the call, even if MINONE or PMN MID was specified. If the application needs to know the message identifiers generated by the queue manager, the application must provide MQPMR records containing the *PRMID* field.

The sending application can also specify a particular value for the message identifier, other than MINONE; 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 *MDCID* field of any report message that it generates about this message (depending on the *MDREP* options)

When the queue manager or a message channel agent generates a report message, it sets the *MDMID* field in the way specified by the *MDREP* field of the original message, either RONMI or ROPMI. Applications that generate report messages should also do this.

For the MQGET call, *MDMID* 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:

MDMID
MDCID
MDGID
MDSEQ
MDOFF

The application sets one or more of these field to the values required, and then sets the corresponding MO* match options in the *GMMO* 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 *GMMO* field (if not altered by the application) is to match both the message identifier and the correlation identifier.

Normally, the message returned is the *first* message on the queue that satisfies the selection criteria. But if GMBRWN is specified, the message returned is the

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 X'49' and X'C1' through X'C9'). However, the application is not prevented from generating identifiers with initial characters in these ranges.

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 16 on page 63 for more information about how selection criteria are used in various situations.

Specifying MINONE as the message identifier has the same effect as *not* specifying MOMSGI, that is, *any* message identifier will match.

This field is ignored if the GMMUC option is specified in the *GMO* parameter on the MQGET call.

On return from an MQGET call, the *MDMID* field is set to the message identifier of the message returned (if any).

The following special value may be used:

MINONE

No message identifier is specified.

The value is binary zero for the length of the field.

This is an input/output field for the MQGET, MQPUT, and MQPUT1 calls. The length of this field is given by LNMID. The initial value of this field is MINONE.

MDCID (24-byte bit string)

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 PMNCID, 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 *MDCID* field in the way specified by the *MDREP* field of the original message, either ROCMTC or ROPCI. Applications which generate report messages should also do this.

For the MQGET call, *MDCID* 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 *MDMID* field for details of how to specify values for this field.

MQMD - Message descriptor

Specifying CINONE as the correlation identifier has the same effect as *not* specifying MOCORI, that is, *any* correlation identifier will match.

If the GMMUC option is specified in the *GMO* parameter on the MQGET call, this field is ignored.

On return from an MQGET call, the *MDCID* field is set to the correlation identifier of the message returned (if any).

The following special values may be used:

CINONE

No correlation identifier is specified.

The value is binary zero for the length of the field.

CINEWS

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 MQGET call, this is an input/output field. For the MQPUT and MQPUT1 calls, this is an input field if PMNCID is *not* specified, and an output field if PMNCID *is* specified. The length of this field is given by LNCID. The initial value of this field is CINONE.

MDBOC (10-digit signed integer)

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 any of the GMBRW* options.

The accuracy of this count is affected by the *HardenGetBackout* local queue attribute; see “Chapter 38. Attributes for local queues and model queues” on page 299.

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.

MDRQ (48-byte character string)

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 MTRPLY and MTRPRT messages. The name is the local name of a queue that is defined on the queue manager identified by *MDRM*. 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 *MDMT* field has the value MTRQST, or if any reports are requested by the *MDREP* 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.

MQMD - Message descriptor

If the *MDRM* field is blank, the local queue manager looks up the *MDRQ* name in its own queue definitions. If a local definition of a remote queue exists with this name, the *MDRQ* 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, *MDRQ* 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 *MDRQ* 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 *MDRQ* field should be set to blanks; 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 “Chapter 43. Attributes for the queue manager” on page 323).

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 LNQN. The initial value of this field is 48 blank characters.

MDRM (48-byte character string)

Name of reply queue manager.

This is the name of the queue manager to which the reply message or report message should be sent. *MDRQ* is the local name of a queue that is defined on this queue manager.

If the *MDRM* field is blank, the local queue manager looks up the *MDRQ* name in its queue definitions. If a local definition of a remote queue exists with this name, the *MDRM* 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 *MDRM* 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 *MDRM* is replaced in the transmitted message. For more information about names, see the *MQSeries Application Programming Guide*.

MQMD - Message descriptor

If a reply-to queue is not required, it is recommended (although this is not checked) that the *MDRM* field should be set to blanks; 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 LNQMN. The initial value of this field is 48 blank characters.

MDUID (12-byte character string)

User identifier.

The fields from *MDUID* to *MDAOD* contain the identity context and origin context of the message. Usually:

- Identity context relates to the application that *originally* put the message
- Origin context relates to the application that *most-recently* put the message.

These two applications can be the same application, but they can also be different applications (for example, when a message is forwarded from one application to another).

Although identity and origin context usually have the meanings described above, the content of both types of context actually depends on the PM* options that are specified when the message is put. As a result, identity context does not necessarily relate to the application that originally put the message, and origin context does not necessarily relate to the application that most-recently put the message — it depends on the design of the application suite.

There is one class of application that never alters message context, namely the message channel agent (MCA). MCAs that receive messages from remote queue managers use the context option PMSETA on the MQPUT or MQPUT1 call. This allows the receiving MCA to preserve exactly the message context that travelled with the message from the sending MCA. However, the result is that the origin context does not relate to the application that most-recently put the message (the receiving MCA), but instead relates to an earlier application that put the message (possibly the originating application itself).

In the descriptions that follow, the context fields are described as though they are used in the normal way. For more information about message context, see the *MQSeries Application Programming Guide*.

MDUID is part of the **identity context** of the message. It specifies the user identifier of the application that originated the message. The queue manager treats this information as character data, but does not define the format of it.

After a message has been received, *MDUID* can be used in the *ODAU* field of the *OBJDSC* parameter of a subsequent MQOPEN or MQPUT1 call, so that the authorization check is performed for the *MDUID* user instead of the application performing the open.

When the queue manager generates this information for an MQPUT or MQPUT1 call:

- On OS/390, the queue manager uses the *ODAU* from the *OBJDSC* parameter of the MQOPEN or MQPUT1 call if the OOALTU or PMALTU option was specified. If the relevant option was not specified, the queue manager uses a user identifier determined from the environment.
- In other environments, the queue manager always uses a user identifier determined from the environment.

When the user identifier is determined from the environment:

- On OS/400, the queue manager uses the name of the user profile associated with the application job.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETI or PMSETA is specified in the *PMO* 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 PMSETI or PMSETA 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 *MDUID* 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 LNUID. The initial value of this field is 12 blank characters.

MDACC (32-byte bit string)
Accounting token.

This is part of the **identity context** of the message. For more information about message context, see the description of the *MDUID* field above; also see the *MQSeries Application Programming Guide*.

MDACC 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 is set as follows:

- The first byte of the field is set to the length of the accounting information present in the bytes that follow; this length is in the range zero through 30, and is stored in the first byte as a binary integer.
- The second and subsequent bytes (as specified by the length field) are set to the accounting information appropriate to the environment.
 - On OS/400, the accounting information is set to the accounting code for the job.
- The last byte is set to the accounting-token type, one of the following values:
 - ATTCIC**
CICS LUOW identifier.
 - ATTDOS**
DOS client default accounting token.
 - ATTWNT**
Windows NT security identifier.
 - ATTOS2**
OS/2 default accounting token.
 - ATT400**
OS/400 accounting token.

MQMD - Message descriptor

ATTUNX

UNIX systems numeric identifier.

ATTWIN

Windows client, 16-bit Windows, or 32-bit Windows default accounting token.

ATTUSR

User-defined accounting token.

ATTUNK

Unknown accounting-token type.

The accounting-token type is set to an explicit value only in the following environments: AIX, DOS client, HP-UX, OS/2, Sun Solaris, Windows client, and Windows NT. In other environments, the accounting-token type is set to the value ATTUNK. In these environments the *MDPAT* field can be used to deduce the type of accounting token received.

- All other bytes are set to binary zero.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETI or PMSETA is specified in the *PMO* parameter. If neither PMSETI nor PMSETA 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 *MDACC* 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 for the *MDACC* field:

ACNONE

No accounting token is specified.

The value is binary zero for the length of the field.

The length of this field is given by LNAACCT. The initial value of this field is ACNONE.

MDAID (32-byte character string)

Application data relating to identity.

This is part of the **identity context** of the message. For more information about message context, see the description of the *MDUID* field above; also see the *MQSeries Application Programming Guide*.

MDAID 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 PMSETI or PMSETA is specified in the *PMO* parameter. If a null character is present, the null and any following characters are converted to blanks by the queue manager. If neither PMSETI nor PMSETA 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 *MDAID* that was transmitted with the message. If the message has no context, the field is entirely blank.

On VSE/ESA, this is a reserved field.

This is an output field for the MQGET call. The length of this field is given by LNAIDD. The initial value of this field is 32 blank characters.

MDPAT (10-digit signed integer)

Type of application that put the message.

This is part of the **origin context** of the message. For more information about message context, see the description of the *MDUID* field above; also see the *MQSeries Application Programming Guide*.

MDPAT may have one of the following standard types. User-defined types can also be used but should be restricted to values in the range ATUFST through ATULST.

ATAIX

AIX application (same value as ATUNIX).

ATCICS

CICS transaction.

ATCICB

CICS bridge.

ATVSE

CICS/VSE transaction.

ATDOS

DOS client application.

ATGUAR

Tandem Guardian application (same value as ATNSK).

ATIMS

IMS application.

ATIMSB

IMS bridge.

ATMVS

OS/390 or TSO application (same value as AT390).

ATNOTE

Lotus Notes Agent application.

MQMD - Message descriptor

ATNSK

Tandem NonStop Kernel application.

ATOS2

OS/2 or Presentation Manager application.

AT390 OS/390 application.

AT400 OS/400 application.

ATQM

Queue-manager-generated message.

ATUNIX

UNIX application.

ATVMS

Digital OpenVMS application.

ATVOS

Stratus VOS application.

ATWIN

Windows client or 16-bit Windows application.

ATWINT

Windows NT or 32-bit Windows application.

ATXCF

XCF.

ATDEF

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.

ATUNK

Unknown application type.

This value can be used to indicate that the application type is unknown, even though other context information is present.

ATUFST

Lowest value for user-defined application type.

ATULST

Highest value for user-defined application type.

The following special value can also occur:

ATNCON

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 PMNOC context option is specified).

When a message is retrieved, *MDPAT* can be tested for this value to decide whether the message has context (it is recommended that *MDPAT* is never set to ATNCON, by an application using PMSETA, if any of the other context fields are nonblank).

MQMD - Message descriptor

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 AT400; the queue manager never uses ATCICS on OS/400.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* parameter. If PMSETA 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 *MDPAT* that was transmitted with the message. If the message has no context, the field is set to ATNCON.

This is an output field for the MQGET call. The initial value of this field is ATNCON.

MDPAN (28-byte character string)

Name of application that put the message.

This is part of the **origin context** of the message. For more information about message context, see the description of the *MDUID* field above; also see the *MQSeries Application Programming Guide*.

The format of the *MDPAN* depends on the value of *MDPAT*.

When this field is set by the queue manager (that is, for all options except PMSETA), it is set to value which is determined by the environment:

- On OS/400, the queue manager uses the fully-qualified job name.

For the MQPUT and MQPUT1 calls, this is an input/output field if PMSETA is specified in the *PMO* 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 PMSETA 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 *MDPAN* 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 LNPNAN. The initial value of this field is 28 blank characters.

MDPD (8-byte character string)

Date when message was put.

This is part of the **origin context** of the message. For more information about message context, see the description of the *MDUID* field above; also 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)

MQMD - Message descriptor

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

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 PMSETA is specified in the *PMO* 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 PMSETA 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 *MDPD* that was transmitted with the message. If the message has no context, the field is entirely blank.

On VSE/ESA, this is a reserved field.

This is an output field for the MQGET call. The length of this field is given by LNPDAT. The initial value of this field is 8 blank characters.

MDPT (8-byte character string)

Time when message was put.

This is part of the **origin context** of the message. For more information about message context, see the description of the *MDUID* field above; also 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 *MDPT*. This happens when leap seconds are inserted into the global time standard.

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

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 PMSETA is specified in the *PMO* 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 PMSETA is not specified, this field is ignored on input and is an output-only field.

MQMD - Message descriptor

After the successful completion of an MQPUT or MQPUT1 call, this field contains the *MDPT* that was transmitted with the message. If the message has no context, the field is entirely blank.

On VSE/ESA, this is a reserved field.

This is an output field for the MQGET call. The length of this field is given by LNPTIM. The initial value of this field is 8 blank characters.

MDAOD (4-byte character string)

Application data relating to origin.

This is part of the **origin context** of the message. For more information about message context, see the description of the *MDUID* field above; also see the *MQSeries Application Programming Guide*.

MDAOD 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.

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 PMSETA is specified in the *PMO* 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 PMSETA 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 *MDAOD* that was transmitted with the message. If the message has no context, the field is entirely blank.

On VSE/ESA, this is a reserved field.

This is an output field for the MQGET call. The length of this field is given by LNAORD. The initial value of this field is 4 blank characters.

The remaining fields are not present if *MDVER* is less than MDVER2.

MDGID (24-byte bit string)

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. *MDGID* is also used if segmentation is allowed for the message. In all of these cases, *MDGID* has a non-null value, and one or more of the following flags is set in the *MDMFL* field:

- MFMIIG
- MFLMIIG
- MFSEG
- MFLSEG
- MFSEGA

If none of these flags is set, *MDGID* has the special null value GINONE.

MQMD - Message descriptor

This field need not be set by the application on the MQPUT or MQGET call if:

- On the MQPUT call, PMLOGO is specified.
- On the MQGET call, MOGRPI 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 *MDGID* 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 PMLOGO 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 PMLOGO is *not* specified, the application should request the queue manager to generate the group identifier, by setting *MDGID* to GINONE 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 PMLOGO 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 31 on page 153. 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 *PRGID* field.

On input to the MQGET call, the queue manager uses the value detailed in Table 16 on page 63. 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:

GINONE

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.

The length of this field is given by LNGID. The initial value of this field is GINONE. This field is not present if *MDVER* is less than MDVER2.

MDSEQ (10-digit signed integer)

Sequence number of logical message within group.

MQMD - Message descriptor

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, PMLOGO is specified.
- On the MQGET call, MOSEQN 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 *MDSEQ* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in Table 31 on page 153. 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 16 on page 63. 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 *MDVER* is less than MDVER2.

MDOFF (10-digit signed integer)

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, PMLOGO is specified.
- On the MQGET call, MOOFFS 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 *MDOFF* is set to an appropriate value.

On input to the MQPUT and MQPUT1 calls, the queue manager uses the value detailed in Table 31 on page 153. 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 *MDOLN* field (provided it is not OLUNDF) 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 16 on page 63. 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 *MDVER* is less than MDVER2.

MQMD - Message descriptor

MDMFL (10-digit signed integer)
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 GMCMPM option on the MQGET call, and supplying a buffer that is big enough to accommodate the complete message. (See “Chapter 8. MQGMO - Get-message options” on page 51 for details of the GMCMPM 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:

MFSEGI

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.

MFSEGA

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. MFSEGA can be set without either MFSEG or MFLSEG being set.

When the queue manager segments a message, the queue manager turns on the MFSEG flag in the copy of the MQMD that is sent with each segment, but does not alter the settings of these flags in the MQMD provided by the application on the MQPUT or MQPUT1 call. For the last segment in the logical message, the queue manager also turns on the MFLSEG flag in the MQMD that is sent with the segment.

Note: Care is needed when messages are put with MFSEGA but without PMLOGO. If the message is:

- Not a segment, and
- Not in a group, and
- Not being forwarded,

the application must remember to reset the *MDGID* field to GINONE 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

MQMD - Message descriptor

messages could inadvertently end up with the same group identifier, which might lead to incorrect processing subsequently. See the descriptions of the *MDGID* field and the *PMLOGO* option for more information about when the *MDGID* 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 *FMSTR* 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
- *FMSTR* 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 *FMSTR* messages into segments that are smaller than 16 bytes (other than the last segment).
- The *MDFMT*, *MDCSI*, and *MDENC* 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 *MDREP* field in the MQMD of segments with *MDOFF* greater than zero are modified as follows:
 - For each report type, if the report option is *RO*D*, 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 *RO**.

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 partway

MQMD - Message descriptor

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 RC2255.

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 GMCMPM 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 FMSTR and the character set is DBCS or mixed SBCS/DBCS, the sending application should itself create and put the segments, specifying MFSEGI 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 *GMSEG* field in MQGMO.

The initial value of this flag is MFSEGI.

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:

MFMI

Message is a member of a group.

MFLMI

Message is the last logical message in a group.

If this flag is set, the queue manager turns on MFMI 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, MFLMI is set, but the *MDSEQ* field has the value one.

MFSE

Message is a segment of a logical message.

When MFSE is specified without MFLSE, 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 RC2253.

MFLSE

Message is the last segment of a logical message.

If this flag is set, the queue manager turns on MFSE 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, MFLSE is set, but the *MDOFF* field has the value zero.

When MFLSE 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 PMLOGO 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 PMLOGO is specified:

- If there is no current group or logical message, all of these flags (and combinations of them) are valid.
- Once MFMI has been specified, it must remain on until MFLMI is specified. The call fails with reason code RC2241 if this condition is not satisfied.
- Once MFSE has been specified, it must remain on until MFLSE is specified. The call fails with reason code RC2242 if this condition is not satisfied.
- Once MFSE has been specified without MFMI, MFMI must remain *off* until after MFLSE has been specified. The call fails with reason code RC2242 if this condition is not satisfied.

Table 31 on page 153 shows the valid combinations of the flags, and the values used for various fields.

MQMD - Message descriptor

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 *GMGST* and *GMSST* fields in MQGMO.

Default flags: The following can be specified to indicate that the message has default attributes:

MFNONE

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. MFNONE 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 *MDMFL* field is partitioned into subfields; for details see “Appendix E. Report options” on page 457.

The initial value of this field is MFNONE. This field is not present if *MDVER* is less than MDVER2.

MDOLN (10-digit signed integer)
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, *MDOLN* 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.

MDOLN should be set by the program which generates the report, but if that program does not set the field, *MDOLN* has the following special value:

OLUNDF

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 RC2252.

- In all other cases, the queue manager ignores the field and uses the value OLUNDF instead.

This is an output field on the MQGET call.

The initial value of this field is OLUNDF. This field is not present if *MDVER* is less than MDVER2.

Initial values and RPG declaration

Table 22. Initial values of fields in MQMD

Field name	Name of constant	Value of constant
<i>MDSID</i>	MDSIDV	'Mdbb' (See note 1)
<i>MDVER</i>	MDVER1	1
<i>MDREP</i>	RONONE	0
<i>MDMT</i>	MTDGRM	8
<i>MDEXP</i>	EIULIM	-1
<i>MDFB</i>	FBNONE	0
<i>MDENC</i>	ENNAT	See note 2
<i>MDCSI</i>	CSQM	0
<i>MDFMT</i>	FMNONE	'bbbbbbbb'
<i>MDPRI</i>	PRQDEF	-1
<i>MDPER</i>	PEQDEF	2
<i>MDMID</i>	MINONE	Nulls
<i>MDCID</i>	CINONE	Nulls
<i>MDBOC</i>	None	0
<i>MDRQ</i>	None	Blanks
<i>MDRM</i>	None	Blanks
<i>MDUID</i>	None	Blanks
<i>MDACC</i>	ACNONE	Nulls
<i>MDAID</i>	None	Blanks
<i>MDPAT</i>	ATNCON	0
<i>MDPAN</i>	None	Blanks
<i>MDPD</i>	None	Blanks
<i>MDPT</i>	None	Blanks
<i>MDAOD</i>	None	Blanks
<i>MDGID</i>	GINONE	Nulls
<i>MDSEQ</i>	None	1
<i>MDOFF</i>	None	0
<i>MDMFL</i>	MFNONE	0
<i>MDOLN</i>	OLUNDF	-1

Notes:

1. The symbol 'b' represents a single blank character.
2. The value of this constant is environment-specific.

MQMD - Message descriptor

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQMD Structure
D*
D* Structure identifier
D MDSID          1      4
D* Structure version number
D MDVER          5      8I 0
D* Options for report messages
D MDREP          9      12I 0
D* Message type
D MDMT           13     16I 0
D* Message lifetime
D MDEXP          17     20I 0
D* Feedback or reason code
D MDFB           21     24I 0
D* Numeric encoding of message data
D MDENC          25     28I 0
D* Character set identifier of message data
D MDCSI          29     32I 0
D* Format name of message data
D MDFMT          33     40
D* Message priority
D MDPRI          41     44I 0
D* Message persistence
D MDPER          45     48I 0
D* Message identifier
D MDMID          49     72
D* Correlation identifier
D MDCID          73     96
D* Backout counter
D MDBOC          97     100I 0
D* Name of reply queue
D MDRQ           101    148
D* Name of reply queue manager
D MDRM           149    196
D* User identifier
D MDUID          197    208
D* Accounting token
D MDACC          209    240
D* Application data relating to identity
D MDAID          241    272
D* Type of application that put the message
D MDPAT          273    276I 0
D* Name of application that put the message
D MDPAN          277    304
D* Date when message was put
D MDPD           305    312
D* Time when message was put
D MDPT           313    320
D* Application data relating to origin
D MDAOD          321    324
D* Group identifier
D MDGID          325    348
D* Sequence number of logical message within group
D MDSEQ          349    352I 0
D* Offset of data in physical message from start of logical message
D MDOFF          353    356I 0
D* Message flags
D MDMFL          357    360I 0
D* Length of original message
D MDOLN          361    364I 0
```

Chapter 11. MQMDE - Message descriptor extension

The following table summarizes the fields in the structure.

Table 23. Fields in MQMDE

Field	Description	Page
MESID	Structure identifier	133
MEVER	Structure version number	133
MELEN	Length of MQMDE structure	134
MEENC	Numeric encoding of data that follows MQMDE	134
MECSI	Character set identifier of data that follows MQMDE	134
MEFMT	Format name of data that follows MQMDE	134
MEFLG	General flags	134
MEGID	Group identifier	134
MESEQ	Sequence number of logical message within group	134
MEOFF	Offset of data in physical message from start of logical message	135
MEMFL	Message flags	135
MEOLN	Length of original message	135

Overview

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.

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 *MDFMT* field in MQMD to FMMDE 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 25 on page 135.

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 24 on page 132.

MQMDE - Message descriptor extension

There is one special case. If the application uses a version-2 MQMD to put a message that is a segment (that is, the MFSEG or MFLSEG flag is set), and the format name in the MQMD is FMDLH, 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.

Table 24. 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

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 "Chapter 43. Attributes for the queue manager" on page 323), while in most cases the latter is given by the value of ENNAT. 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 ENNAT 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 ENNAT 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 - Message descriptor extension

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 queue manager sets the *MDFMT* field in MQMD to the value FMMDE 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 is 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 is usually in the queue manager's character set and encoding. However the MQMDE may be in some other character set and encoding if:

- The MQMDE was treated as data on the MQPUT or MQPUT1 call (see Table 24 on page 132 for the circumstances that can cause this).
- The message was received from a remote queue manager connected by a TCP connection, and the receiving message channel agent (MCA) was not set up correctly (see the *MQSeries Intercommunication* manual for further information).

Note: On OS/2 and Windows NT, applications compiled with Micro Focus COBOL use a value of ENNAT 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 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

MESID (4-byte character string)
Structure identifier.

The value must be:

MESIDV

Identifier for message descriptor extension structure.

The initial value of this field is MESIDV.

MEVER (10-digit signed integer)
Structure version number.

The value must be:

MEVER2

Version-2 message descriptor extension structure.

MQMDE - Message descriptor extension

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

MEVERC

Current version of message descriptor extension structure.

The initial value of this field is MEVER2.

MELEN (10-digit signed integer)

Length of MQMDE structure.

The following value is defined:

MELEN2

Length of version-2 message descriptor extension structure.

The initial value of this field is MELEN2.

MEENC (10-digit signed integer)

Numeric encoding of data that follows MQMDE.

The queue manager does not check the value of this field. See the *MDENC* field described in “Chapter 10. MQMD - Message descriptor” on page 83 for more information about data encodings.

The initial value of this field is ENNAT.

MECSI (10-digit signed integer)

Character-set identifier of data that follows MQMDE.

The queue manager does not check the value of this field.

The initial value of this field is 0.

MEFMT (8-byte character string)

Format name of data that follows MQMDE.

The queue manager does not check the value of this field. See the *MDFMT* field described in “Chapter 10. MQMD - Message descriptor” on page 83 for more information about format names.

The initial value of this field is FMNONE.

MEFLG (10-digit signed integer)

General flags.

The following flag can be specified:

MEFNON

No flags.

The initial value of this field is MEFNON.

MEGID (24-byte bit string)

Group identifier.

See the *MDGID* field described in “Chapter 10. MQMD - Message descriptor” on page 83. The initial value of this field is GINONE.

MESEQ (10-digit signed integer)

Sequence number of logical message within group.

MQMDE - Message descriptor extension

See the *MDSEQ* field described in “Chapter 10. MQMD - Message descriptor” on page 83 . The initial value of this field is 1.

MEOFF (10-digit signed integer)

Offset of data in physical message from start of logical message.

See the *MDOFF* field described in “Chapter 10. MQMD - Message descriptor” on page 83. The initial value of this field is 0.

MEMFL (10-digit signed integer)

Message flags.

See the *MDMFL* field described in “Chapter 10. MQMD - Message descriptor” on page 83. The initial value of this field is MFNONE.

MEOLN (10-digit signed integer)

Length of original message.

See the *MDOLN* field described in “Chapter 10. MQMD - Message descriptor” on page 83. The initial value of this field is OLUNDF.

Initial values and RPG declaration

Table 25. Initial values of fields in MQMDE

Field name	Name of constant	Value of constant
<i>MESID</i>	MESIDV	'MDEb' (See note 1)
<i>MEVER</i>	MEVER2	2
<i>MELEN</i>	MELEN2	72
<i>MEENC</i>	ENNAT	See note 2
<i>MECSI</i>	None	0
<i>MEFMT</i>	FMNONE	'bbbbbbbb'
<i>MEFLG</i>	MEFNON	0
<i>MEGID</i>	GINONE	Nulls
<i>MESEQ</i>	None	1
<i>MEOFF</i>	None	0
<i>MEMFL</i>	MFNONE	0
<i>MEOLN</i>	OLUNDF	-1

Notes:

1. The symbol 'b' represents a single blank character.
2. The value of this constant is environment-specific.

RPG declaration

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQMDE Structure
D*
D* Structure identifier
D MESID          1      4
D* Structure version number
D MEVER          5      8I 0
D* Length of MQMDE structure
D MELEN         9      12I 0
D* Numeric encoding of data that follows MQMDE
D MEENC        13      16I 0
D* Character-set identifier of data that follows MQMDE
D MECSI        17      20I 0
D* Format name of data that follows MQMDE
D MEFMT        21      28
D* General flags
D MEFLG        29      32I 0
D* Group identifier
D MEGID        33      56
D* Sequence number of logical message within group
D MESEQ        57      60I 0
D* Offset of data in physical message from start of logical message
D MEOFF        61      64I 0
D* Message flags
D MEMFL        65      68I 0
D* Length of original message
D MEOLN        69      72I 0
```

Chapter 12. MQOD - Object descriptor

The following table summarizes the fields in the structure.

Table 26. Fields in MQOD

Field	Description	Page
<i>ODSID</i>	Structure identifier	138
<i>ODVER</i>	Structure version number	138
<i>ODOT</i>	Object type	138
<i>ODON</i>	Object name	138
<i>ODMN</i>	Object queue manager name	139
<i>ODDN</i>	Dynamic queue name	140
<i>ODAU</i>	Alternate user identifier	140
Note: The remaining fields are not present if <i>ODVER</i> is less than <i>ODVER2</i> .		
<i>ODREC</i>	Number of object records present	141
<i>ODKDC</i>	Number of local queues opened successfully	141
<i>ODUDC</i>	Number of remote queues opened successfully	141
<i>ODIDC</i>	Number of queues that failed to open	141
<i>ODORO</i>	Offset of first object record from start of MQOD	142
<i>ODRRO</i>	Offset of first response record from start of MQOD	142
<i>ODORP</i>	Address of first object record	143
<i>ODRRP</i>	Address of first response record	143
Note: The remaining fields are not present if <i>ODVER</i> is less than <i>ODVER3</i> .		
<i>ODASI</i>	Alternate security identifier	143
<i>ODRQN</i>	Resolved queue name	144
<i>ODRMN</i>	Resolved queue manager name	144

Overview

The MQOD structure is used to specify an object by name. The following types of object are valid:

- Queue or distribution list
- Process definition
- Queue manager

The current version of MQOD is given by *ODVERC*. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow. The declaration of MQOD provided in the COPY file contains the additional fields, but the initial value provided for the *ODVER* field is *ODVER1*. To use the additional fields, the application must set the version number to *ODVERC*. Applications which are intended to be portable between several environments should use a more-recent version MQOD only if all of those environments support that version.

To open a distribution list, *ODVER* must be *ODVER2* or greater.

MQOD - Object descriptor

This structure is an input/output parameter for the MQOPEN and MQPUT1 calls.

Fields

ODSID (4-byte character string)
Structure identifier.

The value must be:

ODSIDV

Identifier for object descriptor structure.

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

ODVER (10-digit signed integer)
Structure version number.

The value must be one of the following:

ODVER1

Version-1 object descriptor structure.

ODVER2

Version-2 object descriptor structure.

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

ODVER3

Version-3 object descriptor structure.

Fields that exist only in the version-3 structure are identified as such in the descriptions that follow.

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

ODVERC

Current version of object descriptor structure.

This is always an input field. The initial value of this field is ODVER1.

ODOT (10-digit signed integer)
Object type.

Type of object being named in *ODON*. Possible values are:

OTQ Queue.

OTNLST

Namelist.

OTPRO

Process definition.

OTQM

Queue manager.

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

ODON (48-byte character string)
Object name.

MQOD - Object descriptor

This is the local name of the object as defined on the queue manager identified by *ODMN*. The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but may contain trailing blanks. A null character can be used to indicate the end of significant data in the name; the null and any characters following it are treated as blanks.

The following restrictions apply in the environments indicated:

- On systems that use EBCDIC Katakana, lowercase characters cannot be used.
- On OS/400, names containing lowercase characters, forward slash, or percent, must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified in the *QMNAME* parameter.

If *ODOT* is *OTQM*, 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 *ODON* 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 *ODON* 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, *ODREC* is present and greater than zero), *ODON* must be blank or the null string. If this condition is not satisfied, the call fails with reason code *RC2152*.

This is an input/output field for the *MQOPEN* call when *ODON* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by *LNQN*. The initial value of this field is 48 blank characters.

ODMN (48-byte character string)

Object queue manager name.

This is the name of the queue manager on which the *ODON* object is defined. The characters that are valid in the name are the same as those for *ODON* (see above).

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 *ODOT* is *OTNLST*, *OTPRO*, or *OTQM*, the name of the local queue manager must either be specified explicitly, or specified as blank.

If *ODON* 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 *ODMN* 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 *ODON* is the name of a cluster queue, and *ODMN* is blank, the actual destination of messages sent using the queue handle returned by the *MQOPEN* call is chosen by the queue manager (or by a cluster workload exit if there is one):

MQOD - Object descriptor

- If OOBNDO is specified, the queue manager selects a particular instance of the cluster queue during the processing of the MQOPEN call, and all messages put using this queue handle are sent to that instance.
- If OOBNDN is specified, the queue manager may choose a different instance of the destination queue (residing on a different queue manager in the cluster) on each successive MQPUT call that uses this queue handle.

If the application needs to send a message to a *specific* instance of a cluster queue (that is, a queue instance that resides on a particular queue manager), the application should specify the name of that queue manager in the *ODMN* field. This forces the local queue manager to send the message to the specified destination queue manager.

If a distribution list is being opened (that is, *ODREC* is greater than zero), *ODMN* must be blank or the null string. If this condition is not satisfied, the call fails with reason code RC2153.

This is an input/output field for the MQOPEN call when *ODON* is the name of a model queue, and an input-only field in all other cases. The length of this field is given by LNQM. The initial value of this field is 48 blank characters.

ODDN (48-byte character string)
Dynamic queue name.

This is the name of a dynamic queue that is to be created by the MQOPEN call. This is of relevance only when *ODON* specifies the name of a model queue; in all other cases *ODDN* is ignored.

The characters that are valid in the name are the same as those for *ODON* (see above), except that an asterisk is also valid (see below). A name that is completely blank (or one in which only blanks appear before the first null character) is not valid if *ODON* is 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.

This is an input field. The length of this field is given by LNQN. The initial value of this field is 'AMQ.*', padded with blanks.

ODAU (12-byte character string)
Alternate user identifier.

If OOALTU is specified for the MQOPEN call, or PMALTU 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).

MQOD - Object descriptor

If OOALTU or PMALTU 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 OOALTU nor PMALTU is specified, this field is ignored.

This is an input field. The length of this field is given by LNUID. The initial value of this field is 12 blank characters.

The remaining fields are not present if *ODVER* is less than ODVER2.

ODREC (10-digit signed integer)

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 *ODREC* being the number of destination queues in the list. It is valid for a distribution list to contain only one destination.

The value of *ODREC* must not be less than zero, and if it is greater than zero *ODOT* must be OTQ; the call fails with reason code RC2154 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 *ODVER* is less than ODVER2.

ODKDC (10-digit signed integer)

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 *ODVER* is less than ODVER2.

ODUDC (10-digit signed integer)

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 *ODVER* is less than ODVER2.

ODIDC (10-digit signed integer)

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 *CMPCOD* parameter on the MQOPEN or MQPUT1 call is CCOK or CCWARN; it is *not* set if the *CMPCOD* parameter is CCFAIL.

MQOD - Object descriptor

This is an output field. The initial value of this field is 0. This field is not present if *ODVER* is less than *ODVER2*.

ODORO (10-digit signed integer)

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. *ODORO* is used only when a distribution list is being opened. The field is ignored if *ODREC* 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 *ODORO*

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 *ODORO* 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.

- By using the pointer field *ODORP*

In this case, the application can declare the array of MQOR structures separately from the MQOD structure, and set *ODORP* to the address of the array.

Whichever technique is chosen, one of *ODORO* and *ODORP* must be used; the call fails with reason code RC2155 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 *ODVER* is less than *ODVER2*.

ODRRO (10-digit signed integer)

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. *ODRRO* is used only when a distribution list is being opened. The field is ignored if *ODREC* 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 (*RRCC* field in MQRR), and the reason for each failure (*RRREA* 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 RC2136 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 *ODREC* of them.

The response records can be provided in the same way as the object records, either by specifying an offset in *ODRRO*, or by specifying an address in *ODRRP*; see the description of *ODORO* above for details of how to do this. However, no more than one of *ODRRO* and *ODRRP* can be used; the call fails with reason code RC2156 if both are nonzero.

MQOD - Object descriptor

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 CCOK or CCWARN.

This is an input field. The initial value of this field is 0. This field is not present if *ODVER* is less than ODVER2.

ODORP (pointer)

Address of first object record.

This is the address of the first MQOR object record. *ODORP* is used only when a distribution list is being opened. The field is ignored if *ODREC* is zero.

Either *ODORP* or *ODORO* can be used to specify the object records, but not both; see the description of the *ODORO* field above for details. If *ODORP* 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. This field is not present if *ODVER* is less than ODVER2.

ODRRP (pointer)

Address of first response record.

This is the address of the first MQRR response record. *ODRRP* is used only when a distribution list is being opened. The field is ignored if *ODREC* is zero.

Either *ODRRP* or *ODRRO* can be used to specify the response records, but not both; see the description of the *ODRRO* field above for details. If *ODRRP* 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. This field is not present if *ODVER* is less than ODVER2.

The remaining fields are not present if *ODVER* is less than ODVER3.

ODASI (40-byte bit string)

Alternate security identifier.

This is a security identifier that is passed with the *ODAU* to the authorization service to allow appropriate authorization checks to be performed. *ODASI* is used only if:

- OOALTU is specified on the MQOPEN call, or
- PMALTU is specified on the MQPUT1 call,

and the *ODAU* field is not entirely blank up to the first null character or the end of the field.

The *ODASI* field has the following structure:

- The first byte is a binary integer containing the length of the significant data that follows; the length excludes the length byte itself. If no security identifier is present, the length is zero.
- The second byte indicates the type of security identifier that is present; the following values are possible:

MQOD - Object descriptor

SITWNT

Windows NT security identifier.

SITNON

No security identifier.

- The third and subsequent bytes up to the length defined by the first byte contain the security identifier itself.
- Remaining bytes in the field are set to binary zero.

The following special value may be used:

SINONE

No security identifier specified.

The value is binary zero for the length of the field.

This is an input field. The length of this field is given by LNSCID. The initial value of this field is SINONE. This field is not present if *ODVER* is less than ODVER3.

ODRQN (48-byte character string)

Resolved queue name.

This is the name of the final destination queue, as known to the local queue manager. It is set to a nonblank value by the queue manager only for queues that are opened for browse, input, or output (or any combination).

ODRQN is set to blanks if the object opened is any of the following:

- A distribution list
- Not opened for browse, input, or output
- Not a queue

This is an output field. The length of this field is given by LNQN. The initial value of this field is the null string in C, and 48 blank characters in other programming languages. This field is not present if *ODVER* is less than ODVER3.

ODRMN (48-byte character string)

Resolved queue manager name.

This is the name of the final destination queue manager, as known to the local queue manager. It is set to a nonblank value by the queue manager only for queues that are opened for browse, input, or output (or any combination).

ODRMN is set to blanks if the object opened is any of the following:

- A cluster queue with OOBNDN specified (or with OOBNDQ in effect when the *DefBind* queue attribute has the value BNDNOT)
- A distribution list
- Not opened for browse, input, or output
- Not a queue

This is an output field. The length of this field is given by LNQN. The initial value of this field is the null string in C, and 48 blank characters in other programming languages. This field is not present if *ODVER* is less than ODVER3.

Initial values and RPG declaration

Table 27. Initial values of fields in MQOD

Field name	Name of constant	Value of constant
<i>ODSID</i>	ODSIDV	'0Dbb' (See note 1)
<i>ODVER</i>	ODVER1	1
<i>ODOT</i>	OTQ	1
<i>ODON</i>	None	Blanks
<i>ODMN</i>	None	Blanks
<i>ODDN</i>	None	'AMQ.*'
<i>ODAU</i>	None	Blanks
<i>ODREC</i>	None	0
<i>ODKDC</i>	None	0
<i>ODUDC</i>	None	0
<i>ODIDC</i>	None	0
<i>ODORO</i>	None	0
<i>ODRRO</i>	None	0
<i>ODORP</i>	None	Null pointer or null bytes
<i>ODRRP</i>	None	Null pointer or null bytes
<i>ODASI</i>	SINONE	Nulls
<i>ODRQN</i>	None	Blanks
<i>ODRMN</i>	None	Blanks
Notes:		
1. The symbol 'b' represents a single blank character.		

MQOD - Object descriptor

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQOD Structure
D*
D* Structure identifier
D  ODSID          1      4
D* Structure version number
D  ODVER          5      8I 0
D* Object type
D  ODOT           9      12I 0
D* Object name
D  ODON          13      60
D* Object queue manager name
D  ODMN          61      108
D* Dynamic queue name
D  ODDN          109     156
D* Alternate user identifier
D  ODAU          157     168
D* Number of object records present
D  ODREC         169     172I 0
D* Number of local queues opened successfully
D  ODKDC         173     176I 0
D* Number of remote queues opened successfully
D  ODUDC         177     180I 0
D* Number of queues that failed to open
D  ODIDC         181     184I 0
D* Offset of first object record from start of MQOD
D  ODORO         185     188I 0
D* Offset of first response record from start of MQOD
D  ODRRO         189     192I 0
D* Address of first object record
D  ODORP         193     208*
D* Address of first response record
D  ODRRP         209     224*
D* Alternate security identifier
D  ODASI         225     264
D* Resolved queue name
D  ODRQN         265     312
D* Resolved queue manager name
D  ODRMN         313     360
```

Chapter 13. MQOR - Object record

The following table summarizes the fields in the structure.

Table 28. Fields in MQOR

Field	Description	Page
<i>ORON</i>	Object name	147
<i>ORMN</i>	Object queue manager name	147

Overview

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.

Fields

ORON (48-byte character string)
Object name.

This is the same as the *ODON* 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 48 blank characters.

ORMN (48-byte character string)
Object queue manager name.

This is the same as the *ODMN* field in the MQOD structure (see MQOD for details).

This is always an input field. The initial value of this field is 48 blank characters.

RPG declaration

Initial values and RPG declaration

Table 29. Initial values of fields in MQOR

Field name	Name of constant	Value of constant
<i>ORON</i>	None	Blanks
<i>ORMN</i>	None	Blanks

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..  
D* MQOR Structure  
D*  
D* Object name  
D  ORON                1    48  
D* Object queue manager name  
D  ORMN                49   96
```

Chapter 14. MQPMO - Put message options

The following table summarizes the fields in the structure.

Table 30. Fields in MQPMO

Field	Description	Page
<i>PMSID</i>	Structure identifier	150
<i>PMVER</i>	Structure version number	150
<i>PMOPT</i>	Options that control the action of MQPUT and MQPUT1	150
<i>PMCT</i>	Object handle of input queue	158
<i>PMKDC</i>	Number of messages sent successfully to local queues	158
<i>PMUDC</i>	Number of messages sent successfully to remote queues	159
<i>PMIDC</i>	Number of messages that could not be sent	159
<i>PMRQN</i>	Resolved name of destination queue	159
<i>PMRMN</i>	Resolved name of destination queue manager	159
Note: The remaining fields are not present if <i>PMVER</i> is less than <i>PMVER2</i> .		
<i>PMREC</i>	Number of put message records or response records present	160
<i>PMPRF</i>	Flags indicating which MQPMR fields are present	160
<i>PMPRO</i>	Offset of first put-message record from start of MQPMO	161
<i>PMRRO</i>	Offset of first response record from start of MQPMO	162
<i>PMPRP</i>	Address of first put message record	163
<i>PMRRP</i>	Address of first response record	163

Overview

The current version of MQPMO is given by *PMVERC*. Fields that exist only in the more-recent versions of the structure are identified as such in the descriptions that follow. The declaration of MQPMO provided in the COPY file contains the additional fields, but the initial value provided for the *PMVER* field is *PMVER1*. To use the additional fields, the application must set the version number to *PMVERC*. Applications which are intended to be portable between several environments should use a more-recent version MQPMO only if all of those environments support that version.

The MQPMO structure is an input/output parameter for the MQPUT and MQPUT1 calls.

Fields

PMSID (4-byte character string)
Structure identifier.

The value must be:

PMSIDV

Identifier for put-message options structure.

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

PMVER (10-digit signed integer)
Structure version number.

The value must be one of the following:

PMVER1

Version-1 put-message options structure.

PMVER2

Version-2 put-message options structure.

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:

PMVERC

Current version of put-message options structure.

This is always an input field. The initial value of this field is PMVER1.

PMOPT (10-digit signed integer)
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). Combinations that are not valid are noted; any other combinations are valid.

PMSYP

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 PMNSYP is specified, the put request is not within a unit of work.

PMSYP must *not* be specified with PMNSYP.

PMNSYP

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 PMSYP is specified, the put request is not within a unit of work.

PMNSYP must *not* be specified with PMSYP.

PMNMID

Generate a new message identifier.

This option causes the queue manager to replace the contents of the *MDMID* 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 *PRMID* field in the MQPMR structure for details.

Using this option relieves the application of the need to reset the *MDMID* field to MINONE prior to each MQPUT or MQPUT1 call.

PMNCID

Generate a new correlation identifier.

This option causes the queue manager to replace the contents of the *MDCID* 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 *PRCID* field in the MQPMR structure for details.

PMNCID is useful in situations where the application requires a unique correlation identifier.

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 (*MDMID* 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 (*MDGID* field in MQMD), and the same message sequence number (*MDSEQ* field in MQMD). The segments are distinguished by differing values for the segment offset (*MDOFF* 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

MQPMO - Put-message options

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 (GINONE), 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.

PMLOGO

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 PMLOGO 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 *MDGID*, *MDSEQ*, and *MDOFF* fields in MQMD, as the queue manager sets these to the appropriate values. The application need set only the *MDMFL* 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 MF* flags in *MDMFL* 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 RC2241 or RC2242, 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

MQPMO - Put-message options

message data) specifying MFLMIG and/or MFLSEG as appropriate, before reissuing the MQPUT call to put the message that is not in the group or not a segment.

Table 31 shows the combinations of options and flags that are valid, and the values of the *MDGID*, *MDSEQ*, and *MDOFF* 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 columns in the table have the following meanings:

LOG ORD

A “✓” means that the row applies only when the PMLOGO option is specified.

MIG A “✓” means that the row applies only when the MFMIG or MFLMIG option is specified.

SEG A “✓” means that the row applies only when the MFSEG or MFLSEG option is specified.

A “(✓)” means that the row applies whether or not the MFSEG or MFLSEG option is specified.

SEG OK

A “✓” means that the row applies only when the MFSEGA option is specified.

A “(✓)” means that the row applies whether or not the MFSEGA option is specified.

Cur grp

A “✓” means that the row applies only when a current message group exists prior to the call.

A “(✓)” means that the row applies whether or not a current message group exists prior to the call.

Cur log msg

A “✓” means that the row applies only when a current logical message exists prior to the call.

A “(✓)” means that the row applies whether or not a current logical message exists prior to the call.

Other columns

These show the values that the queue manager uses.

“Previous” denotes the value used for the field in the previous message for the queue handle.

Table 31. 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	<i>MDGID</i>	<i>MDSEQ</i>	<i>MDOFF</i>
✓						GINONE	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

MQPMO - Put-message options

Table 31. MQPUT options relating to messages in groups and segments of logical messages (continued)

Options you specify				Group and log-msg status prior to call		Values the queue manager uses		
✓	✓	✓	(✓)	✓	✓	Previous group id	Previous sequence number	Previous offset + previous segment length
				(✓)	(✓)	GINONE	1	0
			✓	(✓)	(✓)	New group id if GINONE, else value in field	1	0
		✓	(✓)	(✓)	(✓)	New group id if GINONE, else value in field	1	Value in field
	✓		(✓)	(✓)	(✓)	New group id if GINONE, else value in field	Value in field	0
	✓	✓	(✓)	(✓)	(✓)	New group id if GINONE, else value in field	Value in field	Value in field

Notes:

- PMLOGO is not valid on the MQPUT1 call.
- For the *MDMID* field, the queue manager generates a new message identifier if *PMNMID* or *MINONE* is specified, and uses the value in the field otherwise.
- For the *MDCID* field, the queue manager generates a new correlation identifier if *PMNCID* is specified, and uses the value in the field otherwise.

When *PMLOGO* 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 *MDPER* 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 *RC2185*.

The *PMLOGO* 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 *RC2245*.

When *PMLOGO* is specified, the *MQMD* supplied on the *MQPUT* call must not be less than *MDVER2*. If this condition is not satisfied, the call fails with reason code *RC2257*.

If *PMLOGO* 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 *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* 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.

MQPMO - Put-message options

When the system restarts, the application can set the *MDGID*, *MDSEQ*, *MDOFF*, *MDMFL*, and *MDPER* fields to the appropriate values, and then issue the MQPUT call with PMSYP or PMNSYP set as desired, but *without* specifying PMLOGO. If this call is successful, the queue manager retains the group and segment information, and subsequent MQPUT calls using that queue handle can specify PMLOGO 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 PMLOGO with MQPUT calls that do not, but the following points should be noted:

- Each successful MQPUT call that does *not* specify PMLOGO 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 PMLOGO 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 CCWARN completion code. Table 32 shows the various cases that can arise. In these cases, if the completion code is not CCOK, the reason code is one of the following (as appropriate):

RC2241
RC2242
RC2185
RC2245

Note: The queue manager does not check the group and segment information for the MQPUT1 call.

Table 32. Outcome when MQPUT or MQCLOSE call not consistent with group and segment information

Current call	Previous call	
	MQPUT with PMLOGO	MQPUT without PMLOGO
MQPUT with PMLOGO	CCFAIL	CCFAIL
MQPUT without PMLOGO	CCWARN	CCOK
MQCLOSE with an unterminated group or logical message	CCWARN	CCOK

Applications that simply want to put messages and segments in logical order are recommended to specify PMLOGO, 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 PMLOGO option, and this can be achieved by not specifying that option. If this is done, the application must ensure that the *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* fields in MQMD are set correctly, prior to each MQPUT or MQPUT1 call.

MQPMO - Put-message options

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 PMLOGO. There are two reasons for this:

- If the messages are retrieved and put in order, specifying PMLOGO 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 PMLOGO, nor the corresponding GMLOGO 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 PMLOGO when putting the report message.

PMLOGO can be specified with any of the other PM* options.

PMNOC

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

PMDEFC

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

<i>MDUID</i>	Determined from the environment if possible; set to blanks otherwise.
<i>MDACC</i>	Determined from the environment if possible; set to ACNONE otherwise.
<i>MDAID</i>	Set to blanks.
<i>MDPAT</i>	Determined from the environment.
<i>MDPAN</i>	Determined from the environment if possible; set to blanks otherwise.
<i>MDPD</i>	Set to date when message is put.
<i>MDPT</i>	Set to time when message is put.
<i>MDAOD</i>	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.

PMPASI

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 *PMCT* field. Origin context information is generated by the queue manager in the same way that it is for PMDEFC (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 OOPASI 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 OOPASI option.

PMPASA

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 *PMCT* 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 OOPASA 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 OOPASA option.

PMSETI

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 PMDEFC (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 OOSETI 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 OOSETI option.

PMSETA

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 OOSETA option. For the MQPUT1 call, the same authorization check is carried out as for the MQOPEN call with the OOSETA option.

Only one of the PM* context options can be specified. If none of these options is specified, PMDEFC is assumed.

PMALTU

Validate with specified user identifier.

MQPMO - Put-message options

This indicates that the *ODAU* field in the *OBJDSC* 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 *ODAU* 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.

PMFIQ

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.

The call returns completion code CCFAIL with reason code RC2161.

PMNONE

No options specified.

This value can be used to indicate that no other options have been specified; all options assume their default values. PMNONE 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 *PMOPT* field is PMNONE.

PMTO (10-digit signed integer)

Reserved.

This is a reserved field; its value is not significant. The initial value of this field is -1.

PMCT (10-digit signed integer)

Object handle of input queue.

If PMPASI or PMPASA 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 PMPASI nor PMPASA is specified, this field is ignored.

This is an input field. The initial value of this field is 0.

PMKDC (10-digit signed integer)

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 *PMVER* is less than PMVER2.

PMUDC (10-digit signed integer)

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 *PMVER* is less than *PMVER2*.

PMIDC (10-digit signed integer)

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 *CMPCOD* parameter on the MQPUT or MQPUT1 call is *CCOK* or *CCWARN*; it is *not* set if the *CMPCOD* parameter is *CCFAIL*.

This is an output field. The initial value of this field is 0. This field is not set if *PMVER* is less than *PMVER2*.

PMRQN (48-byte character string)

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 *PMRMN*.

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 *LNQN*. The initial value of this field is 48 blank characters.

PMRMN (48-byte character string)

Resolved name of destination queue manager.

This is the name of the queue manager (after alias resolution) that owns the queue specified by *PMRQN*.

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 *LNQMN*. The initial value of this field is 48 blank characters.

MQPMO - Put-message options

The remaining fields are not present if *PMVER* is less than *PMVER2*.

PMREC (10-digit signed integer)

Number of put message records or response records present.

This is the number of MQPMPR 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 *PMREC* records of each type.

The value of *PMREC* 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 *PMPRO* below).

If *PMREC* 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 RC2154.

This is an input field. The initial value of this field is 0. This field is not present if *PMVER* is less than *PMVER2*.

PMPRF (10-digit signed integer)

Flags indicating which MQPMPR fields are present.

This field contains flags that must be set to indicate which MQPMPR fields are present in the put message records provided by the application. *PMPRF* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero, or both *PMPRO* and *PMPRP* 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:

PFMID

Message-identifier field is present.

PFCID

Correlation-identifier field is present.

PFGID

Group-identifier field is present.

PFFB Feedback field is present.

PFACC

Accounting-token field is present.

If this flag is specified, either *PMSETI* or *PMSETA* must be specified in the *PMOPT* field; if this condition is not satisfied, the call fails with reason code RC2158.

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

PFNONE

No put-message record fields are present.

If this value is specified, either *PMREC* must be zero, or both *PMPRO* and *PMPRP* must be zero.

PFNONE 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 *PMPRF* contains flags which are not valid, or put message records are provided but *PMPRF* has the value PFNONE, the call fails with reason code RC2158.

This is an input field. The initial value of this field is PFNONE. This field is not present if *PMVER* is less than PMVER2.

PMPRO (10-digit signed integer)

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. *PMPRO* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* 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 *PMREC* 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 *PMPRO*, or by specifying an address

MQPMO - Put-message options

in *PMRRP*; for details of how to do this, see the *ODORO* field described in “Chapter 12. MQOD - Object descriptor” on page 137.

No more than one of *PMPRO* and *PMRRP* can be used; the call fails with reason code RC2159 if both are nonzero.

This is an input field. The initial value of this field is 0. This field is not present if *PMVER* is less than *PMVER2*.

PMRRO (10-digit signed integer)

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. *PMRRO* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* 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 (*RRCC* field in MQRR), and the reason for each failure (*RRREA* 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 RC2136 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 *PMREC* of them.

The response records can be provided in a similar way to the object records in MQOD, either by specifying an offset in *PMRRO*, or by specifying an address in *PMRRP*; for details of how to do this, see the *ODORO* field described in “Chapter 12. MQOD - Object descriptor” on page 137. However, no more than one of *PMRRO* and *PMRRP* can be used; the call fails with reason code RC2156 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.

MQPMO - Put-message options

This is an input field. The initial value of this field is 0. This field is not present if *PMVER* is less than *PMVER2*.

PMPRP (pointer)

Address of first put message record.

This is the address of the first MQPMR put message record. *PMPRP* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero.

Either *PMPRP* or *PMPRO* can be used to specify the put message records, but not both; see the description of the *PMPRO* field above for details. If *PMPRP* 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. This field is not present if *PMVER* is less than *PMVER2*.

PMRRP (pointer)

Address of first response record.

This is the address of the first MQRR response record. *PMRRP* is used only when the message is being put to a distribution list. The field is ignored if *PMREC* is zero.

Either *PMRRP* or *PMRRO* can be used to specify the response records, but not both; see the description of the *PMRRO* field above for details. If *PMRRP* 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. This field is not present if *PMVER* is less than *PMVER2*.

Initial values and RPG declaration

Table 33. Initial values of fields in MQPMO

Field name	Name of constant	Value of constant
<i>PMSID</i>	PMSIDV	'PMOb' (See note 1)
<i>PMVER</i>	PMVER1	1
<i>PMOPT</i>	PMNONE	0
<i>PMTO</i>	None	-1
<i>PMCT</i>	None	0
<i>PMKDC</i>	None	0
<i>PMUDC</i>	None	0
<i>PMIDC</i>	None	0
<i>PMRQN</i>	None	Blanks
<i>PMRMN</i>	None	Blanks
<i>PMREC</i>	None	0
<i>PMPRF</i>	PFNONE	0
<i>PMPRO</i>	None	0

RPG declaration

Table 33. Initial values of fields in MQPMO (continued)

Field name	Name of constant	Value of constant
PMRRO	None	0
PMPRP	None	Null pointer or null bytes
PMRRP	None	Null pointer or null bytes

Notes:

- The symbol 'b' represents a single blank character.

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQPMO Structure
D*
D* Structure identifier
D PMSID          1      4
D* Structure version number
D PMVER          5      8I 0
D* Options that control the action of MQPUT and MQPUT1
D PMOPT          9      12I 0
D* Reserved
D PMTO           13     16I 0
D* Object handle of input queue
D PMCT           17     20I 0
D* Number of messages sent successfully to local queues
D PMKDC          21     24I 0
D* Number of messages sent successfully to remote queues
D PMUDC          25     28I 0
D* Number of messages that could not be sent
D PMIDC          29     32I 0
D* Resolved name of destination queue
D PMRQN          33     80
D* Resolved name of destination queue manager
D PMRMN          81     128
D* Number of put message records or response records present
D PMREC          129    132I 0
D* Flags indicating which MQPMR fields are present
D PMPRF          133    136I 0
D* Offset of first put message record from start of MQPMO
D PMPRO          137    140I 0
D* Offset of first response record from start of MQPMO
D PMRRO          141    144I 0
D* Address of first put message record
D PMPRP          145    160*
D* Address of first response record
D PMRRP          161    176*

```

Chapter 15. MQPMR - Put-message record

The following table summarizes the fields in the structure.

Table 34. Fields in MQPMR

Field	Description	Page
<i>PRMID</i>	Message identifier	165
<i>PRCID</i>	Correlation identifier	166
<i>PRGID</i>	Group identifier	166
<i>PRFB</i>	Feedback or reason code	166
<i>PRACC</i>	Accounting token	167

Overview

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 *PMPRF* 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 COPY file. The application programmer should create a declaration containing the fields that are required by the application, and set the flags in *PMPRF* to indicate the fields that are present.

MQPMR is an input/output structure for the MQPUT and MQPUT1 calls.

Fields

PRMID (24-byte bit string)
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 *MDMID* 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 *PRMID* field. If that value is MINONE, a new message identifier is generated for *each* of those destinations (that is, no two of those destinations have the same message identifier).

MQPMR - Put-message record

If PMNMID 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 PMNCID is processed (see below).

This is an input/output field.

PRCID (24-byte bit string)
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 *MDCID* 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 *PRCID* field.

If PMNCID 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 PMNMID is processed (see above).

This is an input/output field.

PRGID (24-byte bit string)
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 *MDGID* 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 *PRGID* field. The value is processed as documented in Table 31 on page 153, 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 RC2258.

This is an input/output field.

PRFB (10-digit signed integer)
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 *MDFB* 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.

PRACC (32-byte bit string)
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 *MDACC* field in MQMD for a put to a single queue. See the description of *MDACC* in “Chapter 10. MQMD - Message descriptor” on page 83 for information about the content of this field.

If this field is not present, the value in MQMD is used.

This is an input field.

Initial values and RPG declaration

There are no initial values defined for this structure, as no structure declaration is provided. The sample declaration below shows how the structure should be declared by the application programmer if all of the fields are required.

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQPMR Structure
D*
D* Message identifier
D  PRMID                1      24
D* Correlation identifier
D  PRCID                25     48
D* Group identifier
D  PRGID                49     72
D* Feedback or reason code
D  PRFB                73     76I 0
D* Accounting token
D  PRACC               77     108
```

RPG declaration

Chapter 16. MQRMH - Message reference header

The following table summarizes the fields in the structure.

Table 35. Fields in MQRMH

Field	Description	Page
<i>RMSID</i>	Structure identifier	170
<i>RMVER</i>	Structure version number	170
<i>RMLEN</i>	Total length of MQRMH, including strings at end of fixed fields, but not the bulk data	171
<i>RMENC</i>	Numeric encoding of bulk data	171
<i>RMCSI</i>	Character set identifier of bulk data	171
<i>RMFMT</i>	Format name of bulk data	171
<i>RMFLG</i>	Reference message flags	171
<i>RMOT</i>	Object type	172
<i>RMOII</i>	Object instance identifier	172
<i>RMSEL</i>	Length of source environment data	172
<i>RMSEO</i>	Offset of source environment data	172
<i>RMSNL</i>	Length of source object name	173
<i>RMSNO</i>	Offset of source object name	173
<i>RMDEL</i>	Length of destination environment data	173
<i>RMDEO</i>	Offset of destination environment data	173
<i>RMDNL</i>	Length of destination object name	173
<i>RMDNO</i>	Offset of destination object name	174
<i>RMDL</i>	Length of bulk data	174
<i>RMDO</i>	Low offset of bulk data	174
<i>RMDO2</i>	High offset of bulk data	175

Overview

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.

MQRMH - Message reference header

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 *RMLEN* 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 *RMFMT* field in MQRMH has a value other than FMNONE.
- 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 GMCONV option.

The format name of an MQRMH structure is FMRMH. The fields in the MQRMH structure, and the strings addressed by the offset fields, are in the character set and encoding given by the *MDCSI* and *MDENC* 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.

Fields

RMSID (4-byte character string)

Structure identifier.

The value must be:

RMSIDV

Identifier for reference message header structure.

The initial value of this field is RMSIDV.

RMVER (10-digit signed integer)

Structure version number.

The value must be:

RMVER1

Version-1 reference message header structure.

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

RMVERC

Current version of reference message header structure.

The initial value of this field is RMVER1.

RMLen (10-digit signed integer)

Total length of MQRMH, including strings at end of fixed fields, but not the bulk data.

The initial value of this field is zero.

RMENC (10-digit signed integer)

Numeric encoding of bulk data.

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 ENNAT.

RMCSI (10-digit signed integer)

Character set identifier of bulk data.

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 "Chapter 43. Attributes for the queue manager" on page 323 for details of this attribute.

The initial value of this field is 0.

RMFMT (8-byte character string)

Format name of bulk data.

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 FMNONE.

RMFLG (10-digit signed integer)

Reference message flags.

The following flags are defined:

RMLAST

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.

MQRMH - Message reference header

RMNLST

Reference message does not contain or represent last part of object.

RMNLST 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 RMNLST.

RMOT (8-byte character string)

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 *RMFMT* field described above.

The initial value of this field is 8 blanks.

RMOII (24-byte bit string)

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:

OIINON

No object instance identifier specified.

The value is binary zero for the length of the field.

The length of this field is given by LNOIID. The initial value of this field is OIINON.

RMSEL (10-digit signed integer)

Length of source environment data.

If this field is zero, there is no source environment data, and *RMSEO* is ignored.

The initial value of this field is 0.

RMSEO (10-digit signed integer)

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 *RMSEL*; if this length is zero, there is no source environment data, and *RMSEO* is ignored. If present, the source environment data must reside completely within *RMLEN* 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 *RMSNO*, *RMDEO*, and *RMDNO* fields.

MQRMH - Message reference header

The initial value of this field is 0.

RMSNL (10-digit signed integer)
Length of source object name.

If this field is zero, there is no source object name, and *RMSNO* is ignored.

The initial value of this field is 0.

RMSNO (10-digit signed integer)
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 *RMSNL*; if this length is zero, there is no source object name, and *RMSNO* is ignored. If present, the source object name must reside completely within *RMLLEN* 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 *RMSEO*, *RMDEO*, and *RMDNO* fields.

The initial value of this field is 0.

RMDEL (10-digit signed integer)
Length of destination environment data.

If this field is zero, there is no destination environment data, and *RMDEO* is ignored.

RMDEO (10-digit signed integer)
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 *RMDEL*; if this length is zero, there is no destination environment data, and *RMDEO* is ignored. If present, the destination environment data must reside completely within *RMLLEN* 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 *RMSEO*, *RMSNO*, and *RMDNO* fields.

The initial value of this field is 0.

RMDNL (10-digit signed integer)
Length of destination object name.

MQRMH - Message reference header

If this field is zero, there is no destination object name, and *RMDNO* is ignored.

RMDNO (10-digit signed integer)

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 *RMDNL*; if this length is zero, there is no destination object name, and *RMDNO* is ignored. If present, the destination object name must reside completely within *RMLEN* 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 *RMSEO*, *RMSNO*, and *RMDEO* fields.

The initial value of this field is 0.

RMDL (10-digit signed integer)

Length of bulk data.

The *RMDL* 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 *RMLEN* bytes from the start of the MQRMH structure. The length of the entire message minus *RMLEN* gives the length of the bulk data present.

If data is present in the message, *RMDL* specifies the amount of that data that is relevant. The normal case is for *RMDL* 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 *RMDL*, 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.

RMDO (10-digit signed integer)

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 *RMLEN*.

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:

MQRMH - Message reference header

- *RMDO* 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.
- *RMDO2* 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.

RMDO2 (10-digit signed integer)
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 *RMDO* for details.

The initial value of this field is 0.

Initial values and RPG declaration

Table 36. Initial values of fields in MQRMH

Field name	Name of constant	Value of constant
<i>RMSID</i>	RMSIDV	'RMHb' (See note 1)
<i>RMVER</i>	RMVER1	1
<i>RMLLEN</i>	None	0
<i>RMENC</i>	ENNAT	See note 2
<i>RMCSI</i>	None	0
<i>RMFMT</i>	FMNONE	'bbbbbbbb'
<i>RMFLG</i>	RMNLST	0
<i>RMOT</i>	None	'bbbbbbbb'
<i>RMOII</i>	OIINON	Nulls
<i>RMSEL</i>	None	0
<i>RMSEO</i>	None	0
<i>RMSNL</i>	None	0
<i>RMSNO</i>	None	0
<i>RMDEL</i>	None	0
<i>RMDEO</i>	None	0
<i>RMDNL</i>	None	0
<i>RMDNO</i>	None	0
<i>RMDL</i>	None	0
<i>RMDO</i>	None	0
<i>RMDO2</i>	None	0
Notes:		
1. The symbol 'b' represents a single blank character.		
2. The value of this constant is environment-specific.		

RPG declaration

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQRMH Structure
D*
D* Structure identifier
D RMSID          1      4
D* Structure version number
D RMVER          5      8I 0
D* Total length of MQRMH, including strings at end of fixed fields,
D* but not the bulk data
D RMLen          9     12I 0
D* Numeric encoding of bulk data
D RMENC          13     16I 0
D* Character set identifier of bulk data
D RMCSI          17     20I 0
D* Format name of bulk data
D RMFMT          21     28
D* Reference message flags
D RMFLG          29     32I 0
D* Object type
D RMOT           33     40
D* Object instance identifier
D RMOII          41     64
D* Length of source environment data
D RMSEL          65     68I 0
D* Offset of source environment data
D RMSEO          69     72I 0
D* Length of source object name
D RMSNL          73     76I 0
D* Offset of source object name
D RMSNO          77     80I 0
D* Length of destination environment data
D RMDEL          81     84I 0
D* Offset of destination environment data
D RMDEO          85     88I 0
D* Length of destination object name
D RMDNL          89     92I 0
D* Offset of destination object name
D RMDNO          93     96I 0
D* Length of bulk data
D RMDL           97    100I 0
D* Low offset of bulk data
D RMDO          101    104I 0
D* High offset of bulk data
D RMDO2         105    108I 0
```

Chapter 17. MQRR - Response record

The following table summarizes the fields in the structure.

Table 37. Fields in MQRR

Field	Description	Page
RRCC	Completion code for queue	177
RRREA	Reason code for queue	177

Overview

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 RC2136 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.

Fields

RRCC (10-digit signed integer)
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 CCOK.

RRREA (10-digit signed integer)
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 RCNONE.

Initial values and RPG declaration

Table 38. Initial values of fields in MQRR

Field name	Name of constant	Value of constant
RRCC	CCOK	0
RRREA	RCNONE	0

RPG declaration

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQRR Structure
D*
D* Completion code for queue
D RRCC          1      4I 0
D* Reason code for queue
D RRREA        5      8I 0
```

Chapter 18. MQTM - Trigger message

The following table summarizes the fields in the structure.

Table 39. Fields in MQTM

Field	Description	Page
<i>TMSID</i>	Structure identifier	180
<i>TMVER</i>	Structure version number	180
<i>TMQN</i>	Name of triggered queue	180
<i>TMPN</i>	Name of process object	181
<i>TMTD</i>	Trigger data	181
<i>TMAT</i>	Application type	181
<i>TMAI</i>	Application identifier	182
<i>TMED</i>	Environment data	182
<i>TMUD</i>	User data	182

Overview

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 *TMQN*, *TMTD*, and *TMUD*. 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 “Chapter 19. MQTMC2 - Trigger message (character format)” on page 185.

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

<i>MDSID</i>	MDSIDV
<i>MDVER</i>	MDVER1
<i>MDREP</i>	RONONE
<i>MDMT</i>	MTDGRM
<i>MDEXP</i>	EIULIM
<i>MDFB</i>	FBNONE
<i>MDENC</i>	ENNAT
<i>MDCSI</i>	Queue manager's <i>CodedCharSetId</i> attribute
<i>MDFMT</i>	FMTM
<i>MDPRI</i>	Initiation queue's <i>DefPriority</i> attribute
<i>MDPER</i>	PENPER

MQTM - Trigger message

<i>MDMID</i>	A unique value
<i>MDCID</i>	CINONE
<i>MDBOC</i>	0
<i>MDRQ</i>	Blanks
<i>MDRM</i>	Name of queue manager
<i>MDUID</i>	Blanks
<i>MDACC</i>	ACNONE
<i>MDAID</i>	Blanks
<i>MDPAT</i>	ATQM, or as appropriate for the message channel agent
<i>MDPAN</i>	First 28 bytes of the queue-manager name
<i>MDPD</i>	Date when trigger message is sent
<i>MDPT</i>	Time when trigger message is sent
<i>MDAOD</i>	Blanks

An application that generates a trigger message is recommended to set similar values, except for the following:

- The *MDPRI* field can be set to PRQDEF (the queue manager will change this to the default priority for the initiation queue when the message is put).
- The *MDRM* 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 as appropriate for the application.

Fields

TMSID (4-byte character string)
Structure identifier.

The value must be:

TMSIDV

Identifier for trigger message structure.

The initial value of this field is TMSIDV.

TMVER (10-digit signed integer)
Structure version number.

The value must be:

TMVER1

Version number for trigger message structure.

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

TMVERC

Current version of trigger message structure.

The initial value of this field is TMVER1.

TMQN (48-byte character string)
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 “Chapter 37. Attributes for all queues” on page 293 for details of this attribute.

MQTM - Trigger message

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 LNQN. The initial value of this field is 48 blank characters.

TMPN (48-byte character string)
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 *TMQN* field; see “Chapter 38. Attributes for local queues and model queues” on page 299 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 LNPRON. The initial value of this field is 48 blank characters.

TMTD (64-byte character string)
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 *TMQN* field; see “Chapter 38. Attributes for local queues and model queues” on page 299 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 LNTRGD. The initial value of this field is 64 blank characters.

TMAT (10-digit signed integer)
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 *AppType* attribute of the process object identified by the *TMPN* field; see “Chapter 42. Attributes for process definitions” on page 319 for details of this attribute. The content of this data is of no significance to the queue manager.

TMAT can have one of the following standard values. User-defined types can also be used, but should be restricted to values in the range ATUFST through ATULST:

ATCICS

CICS transaction.

ATVSE

CICS/VSE transaction.

AT400 OS/400 application.

ATUFST

Lowest value for user-defined application type.

ATULST

Highest value for user-defined application type.

MQTM - Trigger message

The initial value of this field is 0.

TMAI (256-byte character string)
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 *AppId* attribute of the process object identified by the *TMPN* field; see “Chapter 42. Attributes for process definitions” on page 319 for details of this attribute. The content of this data is of no significance to the queue manager.

The meaning of *TMAI* is determined by the trigger-monitor application. The trigger monitor provided by MQSeries requires *TMAI* to be the name of an executable program.

The length of this field is given by LNPROA. The initial value of this field is 256 blank characters.

TMED (128-byte character string)
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 *TMPN* field; see “Chapter 42. Attributes for process definitions” on page 319 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 LNPROE. The initial value of this field is 128 blank characters.

TMUD (128-byte character string)
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 *TMPN* field; see “Chapter 42. Attributes for process definitions” on page 319 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 LNPROU. The initial value of this field is 128 blank characters.

Initial values and RPG declaration

Table 40. Initial values of fields in MQTM

Field name	Name of constant	Value of constant
<i>TMSID</i>	TMSIDV	'TMbb' (See note 1)
<i>TMVER</i>	TMVER1	1
<i>TMQN</i>	None	Blanks
<i>TMPN</i>	None	Blanks

Table 40. Initial values of fields in MQTM (continued)

Field name	Name of constant	Value of constant
TMTD	None	Blanks
TMAT	None	0
TMAI	None	Blanks
TMED	None	Blanks
TMUD	None	Blanks
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQTM Structure
D*
D* Structure identifier
D  TMSID          1      4
D* Structure version number
D  TMVER          5      8I 0
D* Name of triggered queue
D  TMQN          9      56
D* Name of process object
D  TMPN         57     104
D* Trigger data
D  TMTD        105     168
D* Application type
D  TMAT        169     172I 0
D* Application identifier
D  TMAI        173     428
D* Environment data
D  TMED        429     556
D* User data
D  TMUD        557     684

```

MQTM - Trigger message

Chapter 19. MQTMC2 - Trigger message (character format)

The following table summarizes the fields in the structure.

Table 41. Fields in MQTMC2

Field	Description	Page
<i>TC2SID</i>	Structure identifier	186
<i>TC2VER</i>	Structure version number	186
<i>TC2QN</i>	Name of triggered queue	186
<i>TC2PN</i>	Name of process object	186
<i>TC2TD</i>	Trigger data	186
<i>TC2AT</i>	Application type	186
<i>TC2AI</i>	Application identifier	186
<i>TC2ED</i>	Environment data	186
<i>TC2UD</i>	User data	186
<i>TC2QMN</i>	Queue manager name	186

Overview

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 *TC2QN*, *TC2TD*, and *TC2UD*. 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 OS/390, for an ATIMS application that is started using the CSQQTRMN application, an MQTMC2 structure is made available to the started application.
- On VSE/ESA, this structure is not supported.
- 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 “Chapter 18. MQTM - Trigger message” on page 179 for details of the fields that are the same in this structure.

Fields

TC2SID (4-byte character string)
Structure identifier.

The value must be:

TCSIDV

Identifier for trigger message (character format) structure.

TC2VER (4-byte character string)
Structure version number.

The value must be:

TCVER2

Version 2 trigger message (character format) structure.

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

TCVERC

Current version of trigger message (character format) structure.

TC2QN (48-byte character string)
Name of triggered queue.

See the *TMQN* field in the MQTM structure.

TC2PN (48-byte character string)
Name of process object.

See the *TMPN* field in the MQTM structure.

TC2TD (64-byte character string)
Trigger data.

See the *TMTD* field in the MQTM structure.

TC2AT (4-byte character string)
Application type.

This field always contains blanks, whatever the value in the *TMAT* field in the MQTM structure of the original trigger message.

TC2AI (256-byte character string)
Application identifier.

See the *TMAI* field in the MQTM structure.

TC2ED (128-byte character string)
Environment data.

See the *TMED* field in the MQTM structure.

TC2UD (128-byte character string)
User data.

See the *TMUD* field in the MQTM structure.

TC2QMN (48-byte character string)
Queue manager name.

MQTMC2 - Trigger message (character format)

This is the name of the queue manager at which the trigger event occurred.

Initial values and RPG declaration

Table 42. Initial values of fields in MQTMC2

Field name	Name of constant	Value of constant
TC2SID	TCSIDV	'TMCb' (See note 1)
TC2VER	TCVER2	'bbb2'
TC2QN	None	Blanks
TC2PN	None	Blanks
TC2TD	None	Blanks
TC2AT	None	'bbbb'
TC2AI	None	Blanks
TC2ED	None	Blanks
TC2UD	None	Blanks
TC2QMN	None	Blanks

Notes:
1. The symbol 'b' represents a single blank character.

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQTMC2 Structure
D*
D* Structure identifier
D TC2SID          1      4
D* Structure version number
D TC2VER          5      8
D* Name of triggered queue
D TC2QN           9     56
D* Name of process object
D TC2PN          57    104
D* Trigger data
D TC2TD         105    168
D* Application type
D TC2AT         169    172
D* Application identifier
D TC2AI         173    428
D* Environment data
D TC2ED         429    556
D* User data
D TC2UD         557    684
D* Queue manager name
D TC2QMN        685    732
```

RPG declaration

Chapter 20. MQWIH - Work information header

The following table summarizes the fields in the structure.

Table 43. Fields in MQWIH

Field	Description	Page
<i>WISID</i>	Structure identifier	189
<i>WIVER</i>	Structure version number	189
<i>WILEN</i>	Length of MQWIH structure	190
<i>WIENC</i>	Numeric encoding of data that follows MQWIH	190
<i>WICSI</i>	Character-set identifier of data that follows MQWIH	190
<i>WIFMT</i>	Format name of data that follows MQWIH	190
<i>WIFLG</i>	Flags	190
<i>WISNM</i>	Service name	190
<i>WISST</i>	Service step name	190
<i>WITOK</i>	Message token	191

Overview

The MQWIH structure describes the information that must be present at the start of a message that is to be handled by the OS/390 workload manager. The format name of this structure is FMWIH.

Fields

WISID (4-byte character string)
Structure identifier.

The value must be:

WISIDV

Identifier for work information header structure.

The initial value of this field is WISIDV.

WIVER (10-digit signed integer)
Structure version number.

The value must be:

WIVER1

Version-1 work information header structure.

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

WIVERC

Current version of work information header structure.

The initial value of this field is WIVER1.

MQWIH - Work information header

WILEN (10-digit signed integer)
Length of MQWIH structure.

The value must be:

WILEN1

Length of version-1 work information header structure.

The following constant specifies the length of the current version:

WILENC

Length of current version of work information header structure.

The initial value of this field is WILEN1.

WIENC (10-digit signed integer)
Numeric encoding of data that follows MQWIH.

This is the numeric encoding of the data that follows the MQWIH structure.
The initial value of this field is 0.

WICSI (10-digit signed integer)
Character-set identifier of data that follows MQWIH.

This is the coded character-set identifier of the data that follows the MQWIH structure. The initial value of this field is 0.

WIFMT (8-byte character string)
Format name of data that follows MQWIH.

This is the MQ format name of the data that follows the MQWIH structure.
The rules for coding this are the same as those for the *MDFMT* field in MQMD.

The length of this field is given by LNFMT. The initial value of this field is FMNONE.

WIFLG (10-digit signed integer)
Flags

The value must be:

WINONE

No flags.

The initial value of this field is WINONE.

WISNM (32-byte character string)
Service name.

This is the name of the service that is to process the message.

The length of this field is given by LNSVNM. The initial value of this field is 32 blank characters.

WISST (8-byte character string)
Service step name.

This is the name of the step of *WISNM* to which the message relates.

MQWIH - Work information header

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

WITOK (16-byte bit string)

Message token.

This is a message token that uniquely identifies the message.

For the MQPUT and MQPUT1 calls, this field is ignored. The length of this field is given by LNMTOK. The initial value of this field is MTKNON.

WIRSV (32-byte character string)

Reserved.

This is a reserved field; it must be blank.

Initial values and RPG declaration

Table 44. Initial values of fields in MQWIH

Field name	Name of constant	Value of constant
<i>WISID</i>	WISIDV	'WIHb' (See note 1)
<i>WIVER</i>	WIVER1	1
<i>WILEN</i>	WILEN1	120
<i>WIENC</i>	None	0
<i>WICSI</i>	None	0
<i>WIFMT</i>	FMNONE	'bbbbbbbb'
<i>WIFLG</i>	WINONE	0
<i>WISNM</i>	None	Blanks
<i>WISST</i>	None	Blanks
<i>WITOK</i>	MTKNON	Nulls
<i>WIRSV</i>	None	Blanks
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

RPG declaration

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQWIH Structure
D*
D* Structure identifier
D WISID          1      4
D* Structure version number
D WIVER          5      8I 0
D* Length of MQWIH structure
D WILEN          9     12I 0
D* Numeric encoding of data that follows MQWIH
D WIENC         13     16I 0
D* Character-set identifier of data that follows MQWIH
D WICSI         17     20I 0
D* Format name of data that follows MQWIH
D WIFMT         21     28
D* Flags
D WIFLG         29     32I 0
D* Service name
D WISNM         33     64
D* Service step name
D WISST         65     72
D* Message token
D WITOK         73     88
D* Reserved
D WIRSV         89    120
```

Chapter 21. MQXQH - Transmission queue header

The following table summarizes the fields in the structure.

Table 45. Fields in MQXQH

Field	Description	Page
XQSID	Structure identifier	196
XQVER	Structure version number	196
XQRQ	Name of destination queue	196
XQRQM	Name of destination queue manager	196
XQMD	Original message descriptor	196

Overview

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 USTRAN.

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 *MSGDSC* 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 *MSGDSC* 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 *MSGDSC* 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).

MQXQH - Transmission-queue header

The embedded message descriptor is the one that is returned to the application in the *MSGDSC* 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 *MDCSI* and *MDENC* 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 *MSGDSC* parameter of the MQPUT or MQPUT1 call, with the exception of the following:

- The *MDVER* field always has the value MDVER1.
- If the *MDPRI* field has the value PRQDEF, it is replaced by the value of the queue's *DefPriority* attribute.
- If the *MDPER* field has the value PEQDEF, it is replaced by the value of the queue's *DefPersistence* attribute.
- If the *MDMID* field has the value MINONE, or the PMNMID option was specified, or the message is a distribution-list message, *MDMID* 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 *MDMID* field in each of the new embedded message descriptors is the same as that in the original distribution-list message.

- If the PMNCID option was specified, *MDCID* is replaced by a new correlation identifier generated by the queue manager.
- The context fields are set as indicated by the PM* context options specified in the *PMO* parameter; the context fields are the fields *MDUID* through *MDAOD* 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
<i>MDSID</i>	MDSIDV
<i>MDVER</i>	MDVER2
<i>MDREP</i>	Copied from the embedded message descriptor, but with the bits identified by ROAUXM 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.)
<i>MDMT</i>	Copied from the embedded message descriptor.

Field in separate MQMD	Value used
<i>MDEXP</i>	Copied from the embedded message descriptor.
<i>MDFB</i>	Copied from the embedded message descriptor.
<i>MDENC</i>	ENNAT
<i>MDCSI</i>	Queue manager's <i>CodedCharSetId</i> attribute.
<i>MDFMT</i>	FMXQH
<i>MDPRI</i>	Copied from the embedded message descriptor.
<i>MDPER</i>	Copied from the embedded message descriptor.
<i>MDMID</i>	A new value is generated by the queue manager. This message identifier is different from the <i>MDMID</i> that the queue manager may have generated for the embedded message descriptor (see above).
	The <i>MDMID</i> from the embedded message descriptor.
<i>MDCID</i>	0
<i>MDBOC</i>	0
<i>MDRQ</i>	Copied from the embedded message descriptor.
<i>MDRM</i>	Copied from the embedded message descriptor.
<i>MDUID</i>	Copied from the embedded message descriptor.
<i>MDACC</i>	Copied from the embedded message descriptor.
<i>MDAID</i>	Copied from the embedded message descriptor.
<i>MDPAT</i>	ATQM
<i>MDPAN</i>	First 28 bytes of the queue-manager name.
<i>MDPD</i>	Date when message was put on transmission queue.
<i>MDPT</i>	Time when message was put on transmission queue.
<i>MDAOD</i>	Blanks
<i>MDGID</i>	GINONE
<i>MDSEQ</i>	1
<i>MDOFF</i>	0
<i>MDMFL</i>	MFNONE
<i>MDOLN</i>	OLUNDF

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 *MDFMT* field in the *MSGDSC* parameter of the MQPUT or MQPUT1 call must have the value FMXQH.

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 FMXQH being returned in the *MDFMT* field in the *MSGDSC* parameter of the MQGET call. The values returned in the *MDCSI* and *MDENC* fields in the *MSGDSC* 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 *MDCSI* and *MDENC* fields in the embedded message descriptor.

Fields

XQSID (4-byte character string)
Structure identifier.

The value must be:

XQSIDV

Identifier for transmission-queue header structure.

The initial value of this field is XQSIDV.

XQVER (10-digit signed integer)
Structure version number.

The value must be:

XQVER1

Version number for transmission-queue header structure.

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

XQVERC

Current version of transmission-queue header structure.

The initial value of this field is XQVER1.

XQRQ (48-byte character string)
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 *XQRQM* to be a local definition of another remote queue).

If the message is a distribution-list message (that is, the *MDFMT* field in the embedded message descriptor is FMDH), *XQRQ* is blank.

The length of this field is given by LNQN. The initial value of this field is 48 blank characters.

XQRQM (48-byte character string)
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, *XQRQM* is blank.

The length of this field is given by LNQM. The initial value of this field is 48 blank characters.

XQMD (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 *MSGDSC* parameter on the MQPUT or MQPUT1 call when the message was originally put to the remote queue.

MQXQH - Transmission-queue header

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.

Initial values and RPG declaration

Table 46. Initial values of fields in MQXQH

Field name	Name of constant	Value of constant
XQSID	XQSIDV	'XQHb' (See note 1)
XQVER	XQVER1	1
XQRQ	None	Blanks
XQRQM	None	Blanks
XQMD	Same names and values as for MQMD; see Table 22 on page 129	
Notes:		
1. The symbol 'b' represents a single blank character.		

RPG declaration

```

D*..1.....2.....3.....4.....5.....6.....7..
D* MQXQH Structure
D*
D* Structure identifier
D XQSID 1 4
D* Structure version number
D XQVER 5 8I 0
D* Name of destination queue
D XQRQ 9 56
D* Name of destination queue manager
D XQRQM 57 104
D* Original message descriptor
D*
D* Structure identifier
D XQ1SID 105 108
D* Structure version number
D XQ1VER 109 112I 0
D* Report options
D XQ1REP 113 116I 0
D* Message type
D XQ1MT 117 120I 0
D* Expiry time
D XQ1EXP 121 124I 0
D* Feedback or reason code
D XQ1FB 125 128I 0
D* Numeric encoding of message data
D XQ1ENC 129 132I 0
D* Character set identifier of message data
D XQ1CSI 133 136I 0
D* Format name of message data
D XQ1FMT 137 144
D* Message priority
D XQ1PRI 145 148I 0
D* Message persistence
D XQ1PER 149 152I 0
D* Message identifier
D XQ1MID 153 176
D* Correlation identifier
D XQ1CID 177 200

```

RPG declaration

```
D*      Backout counter
D XQ1BOC          201  204I 0
D*      Name of reply-to queue
D XQ1RQ          205  252
D*      Name of reply queue manager
D XQ1RM          253  300
D*      User identifier
D XQ1UID          301  312
D*      Accounting token
D XQ1ACC          313  344
D*      Application data relating to identity
D XQ1AID          345  376
D*      Type of application that put the message
D XQ1PAT          377  380I 0
D*      Name of application that put the message
D XQ1PAN          381  408
D*      Date when message was put
D XQ1PD          409  416
D*      Time when message was put
D XQ1PT          417  424
D*      Application data relating to origin
D XQ1AOD          425  428
```

Part 2. Function calls

Chapter 22. Call descriptions	201	Parameters	241
Conventions used in the call descriptions	201	Usage notes	248
Chapter 23. MQBACK - Back out changes	203	RPG invocation.	250
Syntax.	203	Chapter 32. MQOPEN - Open object	251
Parameters	203	Syntax.	251
Usage notes	204	Parameters	251
RPG invocation.	205	Usage notes	259
Chapter 24. MQBEGIN - Begin unit of work	207	RPG invocation.	264
Syntax.	207	Chapter 33. MQPUT - Put message	265
Parameters	207	Syntax.	265
Usage notes	208	Parameters	265
RPG invocation (ILE).	210	Usage notes	270
Chapter 25. MQCLOSE - Close object	211	RPG invocation.	273
Syntax.	211	Chapter 34. MQPUT1 - Put one message	275
Parameters	211	Syntax.	275
Usage notes	214	Parameters	275
RPG invocation.	215	Usage notes	279
Chapter 26. MQCMIT - Commit changes	217	RPG invocation.	281
Syntax.	217	Chapter 35. MQSET - Set object attributes.	283
Parameters	217	Syntax.	283
Usage notes	218	Parameters	283
RPG invocation.	219	Usage notes	286
Chapter 27. MQCONN - Connect queue manager 221		RPG invocation.	287
Syntax.	221		
Parameters	221		
Usage notes	224		
RPG invocation.	225		
Chapter 28. MQCONNX - Connect queue manager (extended)	227		
Syntax.	227		
Parameters	227		
RPG invocation.	228		
Chapter 29. MQDISC - Disconnect queue manager	229		
Syntax.	229		
Parameters	229		
Usage notes	230		
RPG invocation.	230		
Chapter 30. MQGET - Get message	231		
Syntax.	231		
Parameters	231		
Usage notes	235		
RPG invocation.	239		
Chapter 31. MQINQ - Inquire about object attributes	241		
Syntax.	241		

Function calls

Chapter 22. Call descriptions

This chapter describes the MQI calls:

- MQCLOSE – Close object
- MQCONN – Connect to queue manager
- 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

Note: The calls associated with data conversion, MQXCNVC and MQDATACONVEXIT, are in “Appendix F. Data conversion” on page 461.

Conventions used in the call descriptions

For each call, this chapter gives a description of the parameters and usage of the call. This is followed by typical invocations of the call, and typical declarations of its parameters, in the RPG programming language.

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 its direction; for example:

CMPCOD (9-digit decimal integer) — output

There is more information about the structure data types in “Chapter 1. Elementary data types” on page 3.

The direction of the parameter can be:

Input You (the programmer) must provide this parameter.

Output

The call returns this parameter.

Input/output

You must provide this parameter, but it is modified by the call.

There is also a brief description of the purpose of the parameter, together with a list of any values that the parameter can take.

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 “Appendix A. Return codes” on page 357.

Call descriptions

Usage notes

Additional information about the call, describing how to use it and any restrictions on its use.

RPG invocation

Typical invocation of the call, and declaration of its parameters, in RPG.

Other notational conventions are:

Constants

Names of constants are shown in uppercase; for example, OOOOUT.

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.

Chapter 23. 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 OS/400, this call is not supported for applications running in compatibility mode.

Syntax

MQBACK (*HCONN*, *COMCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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.

COMCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *COMCOD*.

If *COMCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *COMCOD* is CCFail:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2101

(2101, X'835') Object damaged.

RC2123

(2123, X'84B') Result of commit or back-out operation is mixed.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

MQBACK — Back out changes

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

See “Appendix A. Return codes” on page 357 for more details.

Usage notes

1. This call can be used only when the queue manager itself coordinates the unit of work. This is a local unit of work, where the changes affect only MQ resources.

In environments where the queue manager does not coordinate the unit of work, the appropriate back-out call must be used instead of MQBACK. The environment may also support an implicit back out caused by the application terminating abnormally.

- On OS/400, this call can be used for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
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 *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* 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 PMLOGO option described in “Chapter 14. MQPMO - Put message options” on page 149, and the GMLOGO option described in “Chapter 8. MQGMO - Get-message options” on page 51.

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 “Chapter 27. MQCONN - Connect queue manager” on page 221 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.

RPG invocation

```
C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQBACK(HCONN : COMCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..
DMQBACK          PR                EXTPROC('MQBACK')
D* Connection handle
D HCONN          10I 0 VALUE
D* Completion code
D COMCOD        10I 0
D* Reason code qualifying COMCOD
D REASON        10I 0
```

RPG invocation

Chapter 24. 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, OS/400, Sun Solaris, Windows NT.

Syntax

MQBEGIN (*HCONN*, *BEGOP*, *CMPCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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.

BEGOP (MQBO) – input/output
Options that control the action of MQBEGIN.

See “Chapter 3. MQBO - Begin options” on page 15 for details.

BEGOP 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.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK
Successful completion.

CCWARN
Warning (partial completion).

CCFAIL
Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:
RCNONE
(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:
RC2121
(2121, X'849') No participating resource managers registered.

RC2122
(2122, X'84A') Participating resource manager not available.

MQBEGIN — Begin unit of work

If *CMPCOD* is *CCFAIL*:

RC2134

(2134, X'856') Begin-options structure not valid.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2012

(2012, X'7DC') Call not valid in environment.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

RC2128

(2128, X'850') Unit of work already started.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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 resource manager participating, and so the queue manager acts as the unit-of-work coordinator.

- To start this type of unit of work, the PMSYP or GMSYP 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 MQBEGIN is used, the unit of work is started but the call completes with CCWARN and reason code RC2121.

- 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 RC2012.

- 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 RC2128 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 RC2012.
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.
5. On OS/400, the three types of unit of work are supported as follows:
 - **Queue-manager-coordinated local units of work** can be used only when a commitment definition does not exist at the job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
 - **Queue-manager-coordinated global units of work** are not supported.
 - **Externally-coordinated global units of work** can be used only when a commitment definition exists at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must have been issued for the job. If this has been done, the OS/400 COMMIT and ROLLBACK operations apply to MQ resources as well as to resources belonging to other participating resource managers.

RPG invocation

RPG invocation (ILE)

```
C*..1.....2.....3.....4.....5.....6.....7..  
C          CALLP      MQBEGIN(HCONN : BEGOP : CMPCOD :  
C                                REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..  
DMQBEGIN      PR          EXTPROC('MQBEGIN')  
D* Connection handle  
D HCONN              10I 0 VALUE  
D* Options that control the action of MQBEGIN  
D BEGOP              12A  
D* Completion code  
D CMPCOD              10I 0  
D* Reason code qualifying CMPCOD  
D REASON              10I 0
```

Chapter 25. MQCLOSE - Close object

The MQCLOSE call relinquishes access to an object, and is the inverse of the MQOPEN call.

Syntax

MQCLOSE (*HCONN*, *HOBJ*, *OPTS*, *CMPCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

HOBJ (10-digit signed integer) – 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:

HOUNUH

Unusable object handle.

OPTS (10-digit signed integer) – input
Options that control the action of MQCLOSE.

The *OPTS* 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 QDPERM (see the *DefinitionType* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299). The close options are summarized in Table 47 on page 212.

One (and only one) of the following must be specified:

CONONE

No optional close processing required.

This *must* be specified for:

- Objects other than queues
- Predefined queues

MQCLOSE — Close object

- 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.

CODEL

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 RC2045, and the object is not deleted.

COPURG

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 RC2045, and the object is not deleted.

Table 47. 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	CONONE	CODEL	COPURG
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

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is **CCOK**:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is **CCWARN**:

RC2241

(2241, X'8C1') Message group not complete.

RC2242

(2242, X'8C2') Logical message not complete.

If *CMPCOD* is **CCFAIL**:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019

(2019, X'7E3') Object handle not valid.

RC2035

(2035, X'7F3') Not authorized for access.

RC2101

(2101, X'835') Object damaged.

RC2045

(2045, X'7FD') Option not valid for object type.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2162

(2162, X'872') Queue manager shutting down.

RC2055

(2055, X'807') Queue contains one or more messages or uncommitted put or get requests.

RC2102

(2102, X'836') Insufficient system resources available.

RC2063

(2063, X'80F') Security error occurred.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

See “Appendix A. Return codes” on page 357 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 CONONE option.
2. The following points apply 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.
 - If the queue was opened with the OOBROW option, the browse cursor is destroyed. If the queue is subsequently reopened with the OOBROW option, a new browse cursor is created (see the OOBROW 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 GMLK option described in “Chapter 8. MQGMO - Get-message options” on page 51).
3. The following points apply if the object being closed is a *dynamic queue* (either permanent or temporary):

- For a dynamic queue, the options CODEL or COPURG can be specified regardless of the options specified on the corresponding MQOPEN call.
- When a dynamic queue is deleted, all MQGET calls with the GMWT option that are outstanding against the queue are canceled and reason code RC2052 is returned. See the GMWT option described in “Chapter 8. MQGMO - Get-message options” on page 51.

After a dynamic queue has been deleted, any call (other than MQCLOSE) that attempts to reference the queue using a previously acquired *HOBJ* handle fails with reason code RC2052.

Be aware that although a deleted queue cannot be accessed by applications, the queue is not removed from the system, and associated resources are not freed, until such time as all handles that reference the queue have been closed, and all units of work that affect the queue have been either committed or backed out.

- When a permanent dynamic queue is deleted, if the *HOBJ* handle specified on the MQCLOSE call is *not* the one that was 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 is authorized to delete the queue. If the OOALTU option was specified on the MQOPEN call, the user identifier checked is the *ODAU*.

This check is not performed if:

- The handle specified is the one returned by the MQOPEN call that created the queue.
 - The queue being deleted is a temporary dynamic queue.
- When a temporary dynamic queue is closed, if the *HOBJ* handle specified on the MQCLOSE call is the one that was returned by the MQOPEN call that created the queue, the queue is deleted. This occurs regardless of the close options specified on the MQCLOSE call. If there are messages on the queue, they are discarded; no report messages are generated.

If there are uncommitted units of work that affect the queue, the queue and its messages are still deleted, but this does not cause the units of work to fail. However, as described above, the resources associated with the units of work are not freed until each of the units of work has been either committed or backed out.

4. The following points apply if the object being closed is a *distribution list*:
- The only valid close option for a distribution list is CONONE; the call fails with reason code RC2046 or RC2045 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 *CMPCOD* 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 *CMPCOD* 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 CCFAIL, 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 *CMPCOD* 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.

Only applications running in compatibility mode can be connected implicitly; other applications must issue the MQCONN or MQCONNX call to connect to the queue manager explicitly.

RPG invocation

```
C*.1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQCLOSE(HCONN : HOBJ : OPTS :
C                               CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..
DMQCLOSE      PR          EXTPROC('MQCLOSE')
D* Connection handle
D HCONN              10I 0 VALUE
D* Object handle
D HOBJ              10I 0
D* Options that control the action of MQCLOSE
D OPTS              10I 0 VALUE
D* Completion code
D CMPCOD              10I 0
D* Reason code qualifying CMPCOD
D REASON              10I 0
```

Usage notes

Chapter 26. 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 OS/400, this call is not supported for applications running in compatibility mode.

Syntax

MQCMIT (*HCONN*, *COMCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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.

COMCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *COMCOD*.

If *COMCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *COMCOD* is CCWARN:

RC2003

(2003, X'7D3') Unit of work encountered fatal error or backed out.

RC2124

(2124, X'84C') Result of commit operation is pending.

If *COMCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2009

(2009, X'7D9') Connection to queue manager lost.

MQCMIT — Commit changes

RC2018

(2018, X'7E2') Connection handle not valid.

RC2101

(2101, X'835') Object damaged.

RC2123

(2123, X'84B') Result of commit or back-out operation is mixed.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

See “Appendix A. Return codes” on page 357 for more details.

Usage notes

1. This call can be used only when the queue manager itself coordinates the unit of work. This is a local unit of work, where the changes affect only MQ resources.

In environments where the queue manager does not coordinate the unit of work, the appropriate commit call must be used instead of MQCMIT. The environment may also support an implicit commit caused by the application terminating normally.

- On OS/400, this call can be used for local units of work coordinated by the queue manager. This means that a commitment definition must not exist at job level, that is, the STRCMTCTL command with the CMTSCOPE(*JOB) parameter must not have been issued for the job.
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 *MDGID*, *MDSEQ*, *MDOFF*, and *MDMFL* 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 PMLOGO option described in “Chapter 14. MQPMO - Put message options” on page 149, and the GMLOGO option described in “Chapter 8. MQGMO - Get-message options” on page 51.

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.
 - If the application *does not* issue the MQDISC call but otherwise ends normally, the unit of work is backed out.
 - 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.

RPG invocation

```
C*.1.....2.....3.....4.....5.....6.....7..
C                                CALLP      MQCMIT(HCONN : COMCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..
DMQCMIT          PR                EXTPROC('MQCMIT')
D* Connection handle
D HCONN                    10I 0 VALUE
D* Completion code
D COMCOD                    10I 0
D* Reason code qualifying COMCOD
D REASON                    10I 0
```

RPG invocation

Chapter 27. 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 OS/400[®], applications running in compatibility mode do not have to issue this call. These 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.

Other applications (that is, applications not running in compatibility mode) must use the MQCONN or MQCONNX call to connect to the queue manager, and the MQDISC call to disconnect from the queue manager. This is the recommended style of programming.

Syntax

MQCONN (*QMNAME*, *HCONN*, *CMPCOD*, *REASON*)

Parameters

QMNAME (48-byte character string) – input
Name of queue manager.

This is the name of the queue manager to which the application wishes to connect. The name can contain the following characters:

- Uppercase alphabetic characters (A through Z)
- Lowercase alphabetic characters (a through z)
- Numeric digits (0 through 9)
- Period (.), forward slash (/), underscore (_), percent (%)

The name must not contain leading or embedded blanks, but may contain trailing blanks. A null character can be used to indicate the end of significant data in the name; the null and any characters following it are treated as blanks.

The following restrictions apply in the environments indicated:

- On OS/400, names containing lowercase characters, forward slash, or percent must be enclosed in quotation marks when specified on commands. These quotation marks must not be specified in the *QMNAME* parameter.

If the name consists entirely of blanks, the name of the *default* queue manager is used.

The name specified for *QMNAME* must be the name of a *connectable* queue manager.

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

MQCONN — Connect queue manager

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.

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 LNQMN.

HCONN (10-digit signed integer) – 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.

MQCONN — Connect queue manager

The scope of the handle is restricted to the smallest unit of parallel processing supported by the platform on which the application is running; the handle is not valid outside the unit of parallel processing from which the MQCONN call was issued.

- On OS/400, the scope of the handle is the job issuing the call.

On OS/400 for applications running in compatibility mode, the value returned is:

HCDEFH

Default connection handle.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2002

(2002, X'7D2') Application already connected.

If *CMPCOD* is CCFAIL:

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2267

(2267, X'8DB') Unable to load cluster workload exit.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2273

(2273, X'8E1') Error processing MQCONN call.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2035

(2035, X'7F3') Not authorized for access.

RC2137

(2137, X'859') Object not opened successfully.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2161

(2161, X'871') Queue manager quiescing.

RC2162

(2162, X'872') Queue manager shutting down.

MQCONN — Connect queue manager

RC2102

(2102, X'836') Insufficient system resources available.

RC2063

(2063, X'80F') Security error occurred.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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. If the queue manager fails while an application is running, the application must issue the MQCONN call again in order to obtain a new connection handle to use on subsequent MQ calls. The application can issue the MQCONN call periodically until the call succeeds.
If an application is not sure whether it is connected to the queue manager, the application can safely issue an MQCONN call in order to obtain a connection handle. If the application is already connected, the handle returned is the same as that returned by the previous MQCONN call, but with completion code CCWARN and reason code RC2002.
4. When the application has finished using MQ calls, the application should use the MQDISC call to disconnect from the queue manager.
5. On OS/400, applications written for previous releases of the queue manager can run without the need for recompilation. This is called *compatibility mode*. This mode of operation provides a compatible run-time environment for applications. It comprises the following:
 - The service program AMQZSTUB residing in the library QMQM.
AMQZSTUB provides the same public interface as previous releases, and has the same signature. This service program can be used to access the MQI through bound procedure calls.
 - The program QMQM residing in the library QMQM.
QMQM provides a means of accessing the MQI through dynamic program calls.
 - Programs MQCLOSE, MQCONN, MQDISC, MQGET, MQINQ, MQOPEN, MQPUT, MQPUT1, and MQSET residing in the library QMQM.
These programs also provide a means of accessing the MQI through dynamic program calls, but with a parameter list that corresponds to the standard descriptions of the MQ calls.

These three interfaces do not include capabilities that were introduced in version 5.1. For example, the MQBACK, MQCMIT, and MQCONN calls are not supported. The support provided by these interfaces is for single-threaded applications only.

Support for the new MQ calls in single-threaded applications, and for all MQ calls in multi-threaded applications, is provided through the service programs LIBMQM and LIBMQM_R respectively.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..
C           CALLP      MQCONN(QMNAME : HCONN : CMPCOD :
C                               REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
DMQCONN      PR           EXTPROC('MQCONN')
D* Name of queue manager
D QMNAME           48A
D* Connection handle
D HCONN           10I 0
D* Completion code
D CMPCOD           10I 0
D* Reason code qualifying CMPCOD
D REASON           10I 0
```

Usage notes

Chapter 28. 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.

- On OS/400, this call is not supported for applications running in compatibility mode.

Syntax

MQCONNX (*QMNAME*, *CNOPT*, *HCONN*, *CMPCOD*, *REASON*)

Parameters

QMNAME (48-byte character string) – input
Name of queue manager.

See the *QMNAME* parameter described in “Chapter 27. MQCONN - Connect queue manager” on page 221 for details.

CNOPT (MQCNO) – input/output
Options that control the action of MQCONNX.

See “Chapter 5. MQCNO - Connect options” on page 31 for details.

HCONN (10-digit signed integer) – output
Connection handle.

See the *HCONN* parameter described in “Chapter 27. MQCONN - Connect queue manager” on page 221 for details.

CMPCOD (10-digit signed integer) – output
Completion code.

See the *CMPCOD* parameter described in “Chapter 27. MQCONN - Connect queue manager” on page 221 for details.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

See the *REASON* parameter described in “Chapter 27. MQCONN - Connect queue manager” on page 221 for details of possible reason codes.

The following additional reason codes can be returned by the MQCONNX call:

If *CMPCOD* is CCFAIL:

RC2278

(2278, X'8E6') Client connection fields not valid.

RC2139

(2139, X'85B') Connect-options structure not valid.

MQCONN — Connect queue manager (extended)

RC2046

(2046, X'7FE') Options not valid or not consistent.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..  
C           CALLP      MQCONN(QMNAME : CNOPT : HCONN :  
C                               CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..  
DMQCONN      PR           EXTPROC('MQCONN')  
D* Name of queue manager  
D QMNAME           48A  
D* Options that control the action of MQCONN  
D CNOPT           32A  
D* Connection handle  
D HCONN           10I 0  
D* Completion code  
D CMPCOD           10I 0  
D* Reason code qualifying CMPCOD  
D REASON           10I 0
```

Chapter 29. 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 for applications running in compatibility mode, this call is not necessary. See “Chapter 27. MQCONN - Connect queue manager” on page 221 for more information.

Syntax

MQDISC (*HCONN*, *CMPCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

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:

HCUNUH

Unusable connection handle.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCFAIL:

MQDISC — Disconnect queue manager

- RC2219**
(2219, X'8AB') MQI call reentered before previous call complete.
- RC2009**
(2009, X'7D9') Connection to queue manager lost.
- RC2018**
(2018, X'7E2') Connection handle not valid.
- RC2058**
(2058, X'80A') Queue manager name not valid or not known.
- RC2059**
(2059, X'80B') Queue manager not available for connection.
- RC2162**
(2162, X'872') Queue manager shutting down.
- RC2102**
(2102, X'836') Insufficient system resources available.
- RC2071**
(2071, X'817') Insufficient storage available.
- RC2195**
(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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 CONONE.
2. On OS/400 for applications running in compatibility mode, this call need not be used; see the MQCONN call for more details.

RPG invocation

```
C*.1.....2.....3.....4.....5.....6.....7..  
C          CALLP      MQDISC(HCONN : CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..  
DMQDISC      PR          EXTPROC('MQDISC')  
D* Connection handle  
D HCONN          10I 0  
D* Completion code  
D CMPCOD          10I 0  
D* Reason code qualifying CMPCOD  
D REASON          10I 0
```

Chapter 30. MQGET - Get message

The MQGET call retrieves a message from a local queue that has been opened using the MQOPEN call.

Syntax

MQGET (*HCONN*, *HOBJ*, *MSGDSC*, *GMO*, *BUFLN*, *BUFFER*, *DATLEN*,
CMPCOD, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

HOBJ (10-digit signed integer) – 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 “Chapter 32. MQOPEN - Open object” on page 251 for details):

OOINPS
OOINPX
OOINPQ
OOBRW

MSGDSC (MQMD) – input/output
Message descriptor.

This structure describes the attributes of the message required, and the attributes of the message retrieved. See “Chapter 10. MQMD - Message descriptor” on page 83 for details.

If *BUFLN* is less than the message length, *MSGDSC* is still filled in by the queue manager, whether or not GMATM is specified on the *GMO* parameter (see the *GMOPT* field described in “Chapter 8. MQGMO - Get-message options” on page 51).

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 FMMDE in the *MDFMT* field in MQMD indicates that an MQMDE is present.

MQGET — Get message

GMO (MQGMO) – input/output
Options that control the action of MQGET.

See “Chapter 8. MQGMO - Get-message options” on page 51 for details.

BUFLen (10-digit signed integer) – 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 (GMATM 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 “Chapter 38. Attributes for local queues and model queues” on page 299.

BUFFER (1-byte bit string×*BUFLen*) – output
Area to contain the message data.

If *BUFLen* is less than the message length, as much of the message as possible is moved into *BUFFER*; this happens whether or not GMATM is specified on the *GMO* parameter (see the *GMOPT* field described in “Chapter 8. MQGMO - Get-message options” on page 51 for more information).

The character set and encoding of the data in *BUFFER* are given (respectively) by the *MDCSI* and *MDENC* fields returned in the *MSGDSC* 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 GMCONV option can be used with a user-written exit to perform the conversion of the message data (see “Chapter 8. MQGMO - Get-message options” on page 51 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 ENNAT, respectively).

If the call fails, the contents of the buffer may still have changed.

DATLEN (10-digit signed integer) – output
Length of the message.

This is the length in bytes of the application data *in the message*. If this is greater than *BUFLen*, only *BUFLen* 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 *BUFLen* is less than the message length, *DATLEN* is still filled in by the queue manager, whether or not GMATM is specified on the *GMO* parameter (see the *GMOPT* field described in “Chapter 8. MQGMO - Get-message options” on page 51 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 GMCONV option is specified, and the converted message data is too long to fit in *BUFFER*, the value returned for *DATLEN* 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 *DATLEN*.

- The value returned by the data-conversion exit, for application-defined formats.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

The reason codes listed below are the ones that the queue manager can return for the *REASON* parameter. If the application specifies the GMCONV 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 *CMPCOD* is CCOK :

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2120

(2120, X'848') Converted data too big for buffer.

RC2190

(2190, X'88E') Converted string too big for field.

RC2150

(2150, X'866') DBCS string not valid.

RC2110

(2110, X'83E') Message format not valid.

RC2243

(2243, X'8C3') Message segments have differing CCSIDs.

RC2244

(2244, X'8C4') Message segments have differing encodings.

RC2209

(2209, X'8A1') No message locked.

RC2119

(2119, X'847') Message data not converted.

RC2145

(2145, X'861') Source buffer parameter not valid.

RC2111

(2111, X'83F') Source coded character set identifier not valid.

RC2113

(2113, X'841') Packed-decimal encoding in message not recognized.

RC2114

(2114, X'842') Floating-point encoding in message not recognized.

RC2112

(2112, X'840') Source integer encoding not recognized.

MQGET — Get message

- RC2143**
(2143, X'85F') Source length parameter not valid.
- RC2146**
(2146, X'862') Target buffer parameter not valid.
- RC2115**
(2115, X'843') Target coded character set identifier not valid.
- RC2117**
(2117, X'845') Packed-decimal encoding specified by receiver not recognized.
- RC2118**
(2118, X'846') Floating-point encoding specified by receiver not recognized.
- RC2116**
(2116, X'844') Target integer encoding not recognized.
- RC2079**
(2079, X'81F') Truncated message returned (processing completed).
- RC2080**
(2080, X'820') Truncated message returned (processing not completed).

If *CMPCOD* is CCFAIL:

- RC2004**
(2004, X'7D4') Buffer parameter not valid.
- RC2005**
(2005, X'7D5') Buffer length parameter not valid.
- RC2219**
(2219, X'8AB') MQI call reentered before previous call complete.
- RC2009**
(2009, X'7D9') Connection to queue manager lost.
- RC2010**
(2010, X'7DA') Data length parameter not valid.
- RC2016**
(2016, X'7E0') Gets inhibited for the queue.
- RC2186**
(2186, X'88A') Get-message options structure not valid.
- RC2018**
(2018, X'7E2') Connection handle not valid.
- RC2019**
(2019, X'7E3') Object handle not valid.
- RC2241**
(2241, X'8C1') Message group not complete.
- RC2242**
(2242, X'8C2') Logical message not complete.
- RC2259**
(2259, X'8D3') Inconsistent browse specification.
- RC2245**
(2245, X'8C5') Inconsistent unit-of-work specification.
- RC2246**
(2246, X'8C6') Message under cursor not valid for retrieval.
- RC2247**
(2247, X'8C7') Match options not valid.
- RC2026**
(2026, X'7EA') Message descriptor not valid.
- RC2250**
(2250, X'8CA') Message sequence number not valid.
- RC2033**
(2033, X'7F1') No message available.

RC2034	(2034, X'7F2') Browse cursor not positioned on message.
RC2036	(2036, X'7F4') Queue not open for browse.
RC2037	(2037, X'7F5') Queue not open for input.
RC2041	(2041, X'7F9') Object definition changed since opened.
RC2101	(2101, X'835') Object damaged.
RC2046	(2046, X'7FE') Options not valid or not consistent.
RC2052	(2052, X'804') Queue has been deleted.
RC2058	(2058, X'80A') Queue manager name not valid or not known.
RC2059	(2059, X'80B') Queue manager not available for connection.
RC2161	(2161, X'871') Queue manager quiescing.
RC2162	(2162, X'872') Queue manager shutting down.
RC2102	(2102, X'836') Insufficient system resources available.
RC2071	(2071, X'817') Insufficient storage available.
RC2024	(2024, X'7E8') No more messages can be handled within current unit of work.
RC2072	(2072, X'818') Syncpoint support not available.
RC2195	(2195, X'893') Unexpected error occurred.
RC2255	(2255, X'8CF') Unit of work not available for the queue manager to use.
RC2090	(2090, X'82A') Wait interval in MQGMO not valid.
RC2256	(2256, X'8D0') Wrong version of MQGMO supplied.
RC2257	(2257, X'8D1') Wrong version of MQMD supplied.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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 GMBRWF or GMBRWN option is specified on the *GMO* parameter (see the *GMOPT* field described in “Chapter 8. MQGMO - Get-message options” on page 51).

If the GMLK option is specified with one of the browse options, the browsed message is locked so that it is visible only to this handle.

Usage notes

If the *GMUNLK* option is specified, a previously-locked message is unlocked. No message is retrieved in this case, and the *MSGDSC*, *BUFLN*, *BUFFER* and *DATLEN* parameters are not checked or altered.

2. If the application issuing the *MQGET* call is running as an MQ client, it is possible for the message retrieved to be lost if during the processing of the *MQGET* call the MQ client terminates abnormally or the client connection is severed. This arises because the surrogate that is running on the queue-manager's platform and which issues the *MQGET* call on the client's behalf cannot detect the loss of the client until the surrogate is about to return the message to the client; this is *after* the message has been removed from the queue. This can occur for both persistent messages and nonpersistent messages. The risk of losing messages in this way can be eliminated by always retrieving messages within units of work (that is, by specifying the *GMSYP* option on the *MQGET* call, and using the *MQCMIT* or *MQBACK* calls to commit or back out the unit of work when processing of the message is complete). If *GMSYP* is specified, and the client terminates abnormally or the connection is severed, the surrogate backs out the unit of work on the queue manager and the message is reinstated on the queue.

In principle, the same situation can arise with applications that are running on the queue-manager's platform, but in this case the window during which a message can be lost is very small. However, as with MQ clients the risk can be eliminated by retrieving the message within a unit of work.

3. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved provided that certain conditions are satisfied. See the usage notes in the description of the *MQPUT* call for details. If the conditions are satisfied, the messages will be presented to the receiving application in the order in which they were sent, provided that:
 - 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 *MDCID* fields in the messages in a sequence to a value that was unique to that sequence of messages.
 - The receiver does not deliberately change the order of retrieval, for example by specifying a particular *MDMID* or *MDCID*.

If the sending application put the messages as a message group, the messages will be presented to the receiving application in the correct order provided that the receiving application specifies the *GMLOGO* option on the *MQGET* call.

For more information about message groups, see:

- *MDMFL* field in *MQMD*
 - *PMLOGO* option in *MQPMO*
 - *GMLOGO* option in *MQGMO*
4. Applications should test for the feedback code *FBQUIT* in the *MDFB* field of the *MSGDSC* parameter. If this value is found, the application should end. See the *MDFB* field described in "Chapter 10. *MQMD* - Message descriptor" on page 83 for more information.
 5. If the queue identified by *HOBJ* was opened with the *OOSAVA* option, and the completion code from the *MQGET* call is *CCOK* or *CCWARN*, the context associated with the queue handle *HOBJ* is set to the context of the message that has been retrieved (unless the *GMBRWF* or *GMBRWN* 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*.

6. If the GMCONV option is included in the *GMO* 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 *MDFMT* field in the control information in the message identifies the structure of the application data, and the *MDCSI* and *MDENC* fields in the control information in the message specify its character-set identifier and encoding.
 - The application issuing the MQGET call specifies in the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter the character-set identifier and encoding to which the application message data should be converted.
 - If the *MDCSI* and *MDENC* values in the control information in the message are identical to those in the *MSGDSC* 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 *MDFMT* 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:
 - FMADMIN
 - FMCICS
 - FMCMD1
 - FMCMD2
 - FMDLH
 - FMDH
 - FMEVNT
 - FMIMS
 - FMIMVS
 - FMMDE
 - FMPCF
 - FMRMH
 - FMSTR
 - FMTM
 - FMXQH
- The format name FMNONE 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 GMCONV is specified on the MQGET call for a message that has a format name of FMNONE, and the character set or encoding of the message differs from that specified in the *MSGDSC* parameter, the message is still returned in the *BUFFER* parameter (assuming no other errors), but the call completes with completion code CCWARN and reason code RC2110.

FMNONE 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

Usage notes

environment-specific additions. User-specified format names should not begin with the letters “MQ”, as such names may conflict with format names supported in the future.

See “Appendix F. Data conversion” on page 461 for details of the data-conversion exit.

User data in the message can be converted between any supported character sets and encodings. However, be aware that if the message contains one or more MQ header structures, the message cannot be converted from or to a character set that has double-byte or multi-byte characters for any of the characters that are valid in queue names. Reason code RC2111 or RC2115 results if this is attempted, and the message is returned unconverted. Unicode character set UCS-2 is an example of such a character set.

On return from MQGET, the following reason code indicates that the message was converted successfully:

RCNONE

The following reason code indicates that the message *may* have been converted successfully; the application should check the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter to find out:

RC2079

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 F. Data conversion” on page 461.

7. For the built-in formats listed above, the queue manager may perform *default conversion* of character strings in the message when the GMCONV 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 CCOK, instead of completing with CCWARN and reason code RC2111 or RC2115.

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 GMCONV.
 - 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 GMCONV option is not specified by the application on the MQGET call.

8. The *BUFFER* parameter shown in the RPG programming example is declared as a string; this restricts the maximum length of the parameter to 256 bytes. If a larger buffer is required, the parameter should be declared instead as a structure, or as a field in a physical file.

Declaring the parameter as a structure increases the maximum length possible to 9999 bytes, while declaring the parameter as a field in a physical file increases the maximum length possible to approximately 32K bytes.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQGET(HCONN : HOBJ : MSGDSC : GMO :
C                               BUFLN : BUFFER : DATLEN :
C                               CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
DMQGET      PR          EXTPROC('MQGET')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object handle
D HOBJ          10I 0 VALUE
D* Message descriptor
D MSGDSC          364A
D* Options that control the action of MQGET
D GMO          100A
D* Length in bytes of the BUFFER area
D BUFLN          10I 0 VALUE
D* Area to contain the message data
D BUFFER          *   VALUE
D* Length of the message
D DATLEN          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0
```

Usage notes

Chapter 31. 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
- Process definition
- Queue manager

Syntax

MQINQ (*HCONN*, *HOBJ*, *SELCNT*, *SELS*, *IACNT*, *INTATR*, *CALEN*,
CHRATR, *CMPCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

HOBJ (10-digit signed integer) – 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 OOINQ option.

SELCNT (10-digit signed integer) – input
Count of selectors.

This is the count of selectors that are supplied in the *SELS* array. It is the number of attributes that are to be returned. Zero is a valid value. The maximum number allowed is 256.

SELS (10-digit signed integer×*SELCNT*) – input
Array of attribute selectors.

This is an array of *SELCNT* 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 CCFAIL and reason code RC2067.

In the special case of queues:

- If the selector is not valid for queues of *any* type, the call fails with completion code CCFAIL and reason code RC2067.

MQINQ — Inquire about object attributes

- If the selector is applicable *only* to queues of type or types other than that of the object, the call succeeds with completion code CCWARN and reason code RC2068.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (IA* selectors) are returned in *INTATR* in the same order in which these selectors occur in *SELS*. Attribute values that correspond to character attribute selectors (CA* selectors) are returned in *CHRATR* in the same order in which those selectors occur. IA* selectors can be interleaved with the CA* 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 IA* selectors reside within the range IAFRST through IALAST, and the CA* selectors within the range CAFRST through CALAST. For each range, the constants IALSTU and CALSTU define the highest value that the queue manager will accept.
2. If all of the IA* selectors occur first, the same element numbers can be used to address corresponding elements in the *SELS* and *INTATR* arrays.

For the CA* selectors in the following descriptions, the constant that defines the length in bytes of the resulting string in *CHRATR* is given in parentheses.

Selectors for queue manager

CAALTD

Date of most-recent alteration (LNDATE).

CAALTT

Time of most-recent alteration (LNTIME).

CACADX

Automatic channel definition exit name (LNEXN).

CACLWD

Data passed to cluster workload exit (LNEXDA).

CACLWX

Name of cluster workload exit (LNEXN).

CACMDQ

System command input queue name (LNQN).

CADLQ

Name of dead-letter queue (LNQN).

CADXQN

Default transmission queue name (LNQN).

CAQMD

Queue manager description (LNQMD).

CAQMID

Queue-manager identifier (LNQMID).

CAQMN

Name of local queue manager (LNQMN).

CARP

Name of cluster for which queue manager provides repository services (LNQMN).

CARPNL

Name of namelist object containing names of clusters for which queue manager provides repository services (LNNLN).

IAAUTE

Control attribute for authority events.

IACAD

Control attribute for automatic channel definition.

MQINQ — Inquire about object attributes

IACADE

Control attribute for automatic channel definition events.

IACLWL

Cluster workload length.

IACCSI

Coded character set identifier.

IACMDL

Command level supported by queue manager.

IADIST

Distribution list support.

IAINHE

Control attribute for inhibit events.

IALCLE

Control attribute for local events.

IAMHND

Maximum number of handles.

IAMLEN

Maximum message length.

IAMPRI

Maximum priority.

IAMUNC

Maximum number of uncommitted messages within a unit of work.

IAPFME

Control attribute for performance events.

IAPLAT

Platform on which the queue manager resides.

IARMTE

Control attribute for remote events.

IASSE Control attribute for start stop events.

IASYNC

Syncpoint availability.

IATRGI

Trigger interval.

Selectors for namelists

CAALTD

Date of most-recent alteration (LNDATE).

CAALTT

Time of most-recent alteration (LNTIME).

CALSTD

Namelist description (LNNLD).

CALSTN

Name of namelist object (LNNLN).

CANAMS

Names in the namelist (LNQN × Number of names in the list).

IANAMC

Number of names in the namelist.

Selectors for all types of queue

If the queue being inquired is a cluster queue, the selectors that are valid depend on how the queue was resolved; see usage note 4 for further details.

CAALTD

Date of most-recent alteration (LNDATE).

MQINQ — Inquire about object attributes

CAALTT	Time of most-recent alteration (LNTIME).
CAQD	Queue description (LNQD).
CAQN	Queue name (LNQN).
IADPER	Default message persistence.
IADPRI	Default message priority.
IAIPUT	Whether put operations are allowed.
IAQTYP	Queue type.

Selectors for local queues

If the queue being inquired is a cluster queue, the selectors that are valid depend on how the queue was resolved; see usage note 4 for further details.

CABRQN	Excessive backout requeue name (LNQN).
CACLN	Cluster name (LNCLUN).
CACLNL	Cluster namelist (LNNLN).
CACRTD	Queue creation date (LNCRTD).
CACRTT	Queue creation time (LNCRTT).
CAINIQ	Initiation queue name (LNQN).
CAPRON	Name of process definition (LNPRON).
CATRGD	Trigger data (LNTRGD).
IABTHR	Backout threshold.
IACDEP	Number of messages on queue.
IADBND	Default binding.
IADINP	Default open-for-input option.
IADEFT	Queue definition type.
IADIST	Distribution list support.
IAHGB	Whether to harden backout count.
IAIGET	Whether get operations are allowed.
IAMLEN	Maximum message length.
IAMDEP	Maximum number of messages allowed on queue.

MQINQ — Inquire about object attributes

IAMDS

Whether message priority is relevant.

IAOIC

Number of MQOPEN calls that have the queue open for input.

IAOOC

Number of MQOPEN calls that have the queue open for output.

IAQDHE

Control attribute for queue depth high events.

IAQDHL

High limit for queue depth.

IAQDLE

Control attribute for queue depth low events.

IAQDLL

Low limit for queue depth.

IAQDME

Control attribute for queue depth max events.

IAQSI

Limit for queue service interval.

IAQSIE

Control attribute for queue service interval events.

IARINT

Queue retention interval.

IASCOP

Queue definition scope.

IASHAR

Whether queue can be shared.

IATRGC

Trigger control.

IATRGD

Trigger depth.

IATRGP

Threshold message priority for triggers.

IATRGT

Trigger type.

IAUSAG

Usage.

Selectors for local definitions of remote queues

CACLN

Cluster name (LNCLUN).

CACLNL

Cluster namelist (LNNLN).

CARQMN

Name of remote queue manager (LNQMN).

CARQN

Name of remote queue as known on remote queue manager (LNQN).

CAXQN

Transmission queue name (LNQN).

IADBND

Default binding.

IASCOP

Queue definition scope.

Selectors for alias queues

CABASQ

Name of queue that alias resolves to (LNQN).

MQINQ — Inquire about object attributes

CACLN
Cluster name (LNCLUN).
CACLNL
Cluster namelist (LNNLN).
IADBND
Default binding.
IAIGET
Whether get operations are allowed.
IASCOP
Queue definition scope.

Selectors for process definitions

CAALTD
Date of most-recent alteration (LNDATE).
CAALTT
Time of most-recent alteration (LNTIME).
CAAPPI
Application identifier (LNPROA).
CAENVVD
Environment data (LNPROE).
CAPROD
Description of process definition (LNPROD).
CAPRON
Name of process definition (LNPRON).
CAUSRD
User data (LNPROU).
IAAPPT
Application type.

IACNT (10-digit signed integer) – input
Count of integer attributes.

This is the number of elements in the *INTATR* array. Zero is a valid value.

If this is at least the number of IA* selectors in the *SELS* parameter, all integer attributes requested are returned.

INTATR (10-digit signed integer×*IACNT*) – output
Array of integer attributes.

This is an array of *IACNT* integer attribute values.

Integer attribute values are returned in the same order as the IA* selectors in the *SELS* parameter. If the array contains more elements than the number of IA* 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 IAVNA is returned for the corresponding element in the *INTATR* array.

CALEN (10-digit signed integer) – input
Length of character attributes buffer.

This is the length in bytes of the *CHRATR* parameter.

This must be at least the sum of the lengths of the requested character attributes (see *SELS*). Zero is a valid value.

MQINQ — Inquire about object attributes

CHRATR (1-byte character string×*CALEN*) – 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 *CALEN* parameter.

Character attributes are returned in the same order as the CA* selectors in the *SELS* parameter. The length of each attribute string is fixed for each attribute (see *SELS*), 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 *CHRATR*.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

- CCOK**
Successful completion.
- CCWARN**
Warning (partial completion).
- CCFAIL**
Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

- RCNONE**
(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

- RC2008**
(2008, X'7D8') Not enough space allowed for character attributes.
- RC2022**
(2022, X'7E6') Not enough space allowed for integer attributes.
- RC2068**
(2068, X'814') Selector not applicable to queue type.

If *CMPCOD* is CCFAIL:

- RC2219**
(2219, X'8AB') MQI call reentered before previous call complete.
- RC2006**
(2006, X'7D6') Length of character attributes not valid.
- RC2007**
(2007, X'7D7') Character attributes string not valid.
- RC2009**
(2009, X'7D9') Connection to queue manager lost.
- RC2018**
(2018, X'7E2') Connection handle not valid.
- RC2019**
(2019, X'7E3') Object handle not valid.

MQINQ — Inquire about object attributes

RC2021

(2021, X'7E5') Count of integer attributes not valid.

RC2023

(2023, X'7E7') Integer attributes array not valid.

RC2038

(2038, X'7F6') Queue not open for inquire.

RC2041

(2041, X'7F9') Object definition changed since opened.

RC2101

(2101, X'835') Object damaged.

RC2052

(2052, X'804') Queue has been deleted.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2162

(2162, X'872') Queue manager shutting down.

RC2102

(2102, X'836') Insufficient system resources available.

RC2065

(2065, X'811') Count of selectors not valid.

RC2067

(2067, X'813') Attribute selector not valid.

RC2066

(2066, X'812') Count of selectors too big.

RC2071

(2071, X'817') Insufficient storage available.

RC2195

(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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. If the object being inquired is an alias queue, the attribute values returned by the MQINQ call are those of the alias queue, and not those of the base queue to which the alias resolves.
4. If the object being inquired is a cluster queue, the attributes that can be inquired depend on how the queue is opened:
 - If the cluster queue is opened for inquire plus one or more of input, browse, or set, there must be a local instance of the cluster queue in order for the open to succeed. In this case the attributes that can be inquired are those valid for local queues.

- If the cluster queue is opened for inquire alone, or inquire and output, only the attributes listed below can be inquired; the *QType* attribute has the value QTCLUS in this case:

CAQD
CAQN
IADBND
IADPER
IADPRI
IAIPUT
IAQTYP

If the cluster queue is opened with no fixed binding (that is, OOBNDN specified on the MQOPEN call, or OOBNDQ specified when the *DefBind* attribute has the value BNDNOT), successive MQINQ calls for the queue may inquire different instances of the cluster queue, although usually all of the instances have the same attribute values.

For more information about cluster queues, refer to the *MQSeries Queue Manager Clusters* book.

5. 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.
6. If more than one of the warning situations arise (see the *CMPCOD* parameter), the reason code returned is the *first* one in the following list that applies:
 - a. RC2068
 - b. RC2022
 - c. RC2008
7. For more information about object attributes, see Chapter 36. Attributes of MQSeries objects.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..  
C          CALLP      MQINQ(HCONN : HOBJ : SELCNT :  
C                               SELS(1) : IACNT : INTATR(1) :  
C                               CALEN : CHRATR : CMPCOD :  
C                               REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..  
DMQINQ      PR          EXTPROC('MQINQ')  
D* Connection handle  
D HCONN          10I 0 VALUE  
D* Object handle  
D HOBJ          10I 0 VALUE  
D* Count of selectors  
D SELCNT        10I 0 VALUE  
D* Array of attribute selectors  
D SELS          10I 0  
D* Count of integer attributes  
D IACNT         10I 0 VALUE  
D* Array of integer attributes  
D INTATR        10I 0  
D* Length of character attributes buffer  
D CALEN         10I 0 VALUE  
D* Character attributes  
D CHRATR          *  VALUE  
D* Completion code  
D CMPCOD        10I 0  
D* Reason code qualifying CMPCOD  
D REASON        10I 0
```

Chapter 32. MQOPEN - Open object

The MQOPEN call establishes access to an object. The following types of object are valid:

- Queue (including distribution lists)
- Namelist
- Process definition
- Queue manager

Syntax

MQOPEN (*HCONN*, *OBJDSC*, *OPTS*, *HOBJ*, *CMPCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

OBJDSC (MQOD) – input/output
Object descriptor.

This is a structure that identifies the object to be opened; see “Chapter 12. MQOD - Object descriptor” on page 137 for details.

If the *ODON* field in the *OBJDSC* 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 *OPTS* 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 *OBJDSC* 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 “Chapter 38. Attributes for local queues and model queues” on page 299). For information about the close options applicable to dynamic queues, see the description of the MQCLOSE call.

OPTS (10-digit signed integer) – input
Options that control the action of MQOPEN.

At least one of the following options must be specified:

- OOBRW
- OOINP* (only one of these)

MQOPEN — Open object

OOINQ
OOOUT
OOSET

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). Combinations that are not valid are noted; all other combinations are valid. Only options that are applicable to the type of object specified by *OBJDSC* are allowed (see Table 48 on page 256).

Access options: The following options control the type of operations that can be performed on the object:

OOINPQ

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 “Chapter 38. Attributes for local queues and model queues” on page 299 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.

OOINPS

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 Ooinps, but fails with reason code RC2042 if the queue is currently open with Ooinpx.

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.

OOINPX

Open queue to get messages with exclusive access.

The queue is opened for use with subsequent MQGET calls. The call fails with reason code RC2042 if the queue is currently open by this or another application for input of any type (Ooinps or Ooinpx).

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 these options:

- 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 QAGETI (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 QANSHR), 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 *ODMN* 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.

OBRW

Open queue to browse messages.

The queue is opened for use with subsequent MQGET calls with one of the following options:

GMBRWF
GMBRWN
GMBRWC

This is allowed even if the queue is currently open for OOINPX. An MQOPEN call with the OBRW option establishes a browse cursor, and positions it logically before the first message on the queue; see the *GMOPT* field described in “Chapter 8. MQGMO - Get-message options” on page 51 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 *ODMN* 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.

OOOUT

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 QAPUTI (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.

OOINQ

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 *ODMN* 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.

OOSET

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 *ODMN* is the name of a local definition of a remote queue; this is true even if the value of the *RemoteQMgrName* attribute in

MQOPEN — Open object

the local definition of a remote queue used for queue-manager aliasing is the name of the local queue manager.

Binding options: The following options apply when the object being opened is a cluster queue; these options control the binding of the queue handle to a particular instance of the cluster queue:

OOBNDO

Bind handle to destination when queue is opened.

This causes the local queue manager to bind the queue handle to a particular instance of the destination queue when the queue is opened. As a result, all messages put using this handle are sent to the same instance of the destination queue, and by the same route.

This option is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

OOBNDN

Do not bind to a specific destination.

This stops the local queue manager binding the queue handle to a particular instance of the destination queue. As a result, successive MQPUT calls using this handle may result in the messages being sent to *different* instances of the destination queue, or being sent to the same instance but by different routes. It also allows the instance selected to be changed subsequently by the local queue manager, by a remote queue manager, or by a message channel agent (MCA), according to network conditions.

Note: Client and server applications which need to exchange a *series* of messages in order to complete a transaction should not use OOBNDN (or OOBNDQ when *DefBind* has the value BNDNOT), because successive messages in the series may be sent to different instances of the server application.

If OOBRW or one of the OOINP* options is specified for a cluster queue, the queue manager is forced to select the local instance of the cluster queue. As a result, the binding of the queue handle is fixed, even if OOBNDN is specified.

If OOINQ is specified with OOBNDN, successive MQINQ calls using that handle may inquire different instances of the cluster queue, although usually all of the instances have the same attribute values.

OOBNDN is valid only for queues, and affects only cluster queues. If specified for a queue that is not a cluster queue, the option is ignored.

OOBNDQ

Use default binding for queue.

This causes the local queue manager to bind the queue handle in the way defined by the *DefBind* queue attribute. The value of this attribute is either BNDOPN or BNDNOT.

OOBNDQ is the default if neither OOBNDO nor OOBNDN is specified.

OOBNDQ is defined to aid program documentation. It is not intended that this option be used with either of the other two bind options, but because its value is zero such use cannot be detected.

Context options: The following options control the processing of message context:

OOSAVA

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 PMPASI and PMPASA options described in “Chapter 14. MQPMO - Put message options” on page 149.

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 GMBRW* browse options does **not** have its context information saved (although the context fields in the *MSGDSC* 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 OOINP* options must be specified.

OOPASI

Allow identity context to be passed.

This allows the PMPASI option to be specified in the *PMO* 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 OOSAVA option. For more information on message context, see the *MQSeries Application Programming Guide*.

The OOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

OOPASA

Allow all context to be passed.

This allows the PMPASA option to be specified in the *PMO* 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 OOSAVA option. For more information on message context, see the *MQSeries Application Programming Guide*.

This option implies OOPASI, which need not therefore be specified. The OOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

OOSSETI

Allow identity context to be set.

This allows the PMSETI option to be specified in the *PMO* parameter when a message is put on a queue; this gives the message the identity context information contained in the *MSGDSC* parameter specified on the

MQOPEN — Open object

MQPUT or MQPUT1 call. For more information on message context, see the *MQSeries Application Programming Guide*.

This option implies OOPASI, which need not therefore be specified. The OOOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

OOSETA

Allow all context to be set.

This allows the PMSETA option to be specified in the *PMO* parameter when a message is put on a queue; this gives the message the identity and origin context information contained in the *MSGDSC* 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:

- OOPASI
- OOPASA
- OOSETI

The OOOOUT option must be specified.

This option is valid for all types of queue, including distribution lists.

Other options: The following options control authorization checking, and what happens when the queue manager is quiescing:

OOALTU

Validate with specified user identifier.

This indicates that the *ODAU* field in the *OBJDSC* parameter contains a user identifier that is to be used to validate this MQOPEN call. The call can succeed only if this *ODAU* 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.

OOFIQ

Fail if queue manager is quiescing.

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

This option is valid for all types of object.

Table 48. Valid MQOPEN options for each queue type

Option	Alias (see note 1)	Local and Model	Remote	Nonlocal Cluster	Distribution list
OOINPQ	✓	✓	—	—	—
OOINPS	✓	✓	—	—	—
OOINPX	✓	✓	—	—	—
OOBRW	✓	✓	—	—	—
OOOUT	✓	✓	✓	✓	✓
OOINQ	✓	✓	see note 2	✓	—
OOSET	✓	✓	see note 2	—	—

Table 48. Valid MQOPEN options for each queue type (continued)

Option	Alias (see note 1)	Local and Model	Remote	Nonlocal Cluster	Distribution list
OOBNDQ (see note 3)	✓	✓	✓	✓	✓
OOBNDN (see note 3)	✓	✓	✓	✓	✓
OOBNDQ (see note 3)	✓	✓	✓	✓	✓
OOSAVA	✓	✓	—	—	—
OOPASI	✓	✓	✓	✓	✓
OOPASA	✓	✓	✓	✓	✓
OOSSETI	✓	✓	✓	✓	✓
OOSSETA	✓	✓	✓	✓	✓
OOALTU	✓	✓	✓	✓	✓
OOFIQ	✓	✓	✓	✓	✓

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.
3. This option can be specified for any queue type, but is ignored if the queue is not a cluster queue.

*HOB*J (10-digit signed integer) – 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 supported by the platform on which the application is running; the handle is not valid outside the unit of parallel processing from which the MQOPEN call was issued:

- On OS/400, the scope of the handle is the job issuing the call.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2136

(2136, X'858') Multiple reason codes returned.

MQOPEN — Open object

If *CMPCOD* is *CCFAIL*:

RC2001

(2001, X'7D1') Alias base queue not a valid type.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2266

(2266, X'8DA') Cluster workload exit failed.

RC2268

(2268, X'8DC') Put calls inhibited for all queues in cluster.

RC2189

(2189, X'88D') Cluster name resolution failed.

RC2269

(2269, X'8DD') Cluster resource error.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2198

(2198, X'896') Default transmission queue not local.

RC2199

(2199, X'897') Default transmission queue usage error.

RC2011

(2011, X'7DB') Name of dynamic queue not valid.

RC2017

(2017, X'7E1') No more handles available.

RC2018

(2018, X'7E2') Connection handle not valid.

RC2019

(2019, X'7E3') Object handle not valid.

RC2194

(2194, X'892') Object name not valid for object type.

RC2035

(2035, X'7F3') Not authorized for access.

RC2100

(2100, X'834') Object already exists.

RC2101

(2101, X'835') Object damaged.

RC2042

(2042, X'7FA') Object already open with conflicting options.

RC2043

(2043, X'7FB') Object type not valid.

RC2044

(2044, X'7FC') Object descriptor structure not valid.

RC2045

(2045, X'7FD') Option not valid for object type.

RC2046

(2046, X'7FE') Options not valid or not consistent.

RC2052

(2052, X'804') Queue has been deleted.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

RC2059

(2059, X'80B') Queue manager not available for connection.

RC2161

(2161, X'871') Queue manager quiescing.

RC2162

(2162, X'872') Queue manager shutting down.

RC2057	(2057, X'809') Queue type not valid.
RC2184	(2184, X'888') Remote queue name not valid.
RC2102	(2102, X'836') Insufficient system resources available.
RC2063	(2063, X'80F') Security error occurred.
RC2188	(2188, X'88C') Call rejected by cluster workload exit.
RC2071	(2071, X'817') Insufficient storage available.
RC2195	(2195, X'893') Unexpected error occurred.
RC2082	(2082, X'822') Unknown alias base queue.
RC2197	(2197, X'895') Unknown default transmission queue.
RC2085	(2085, X'825') Unknown object name.
RC2086	(2086, X'826') Unknown object queue manager.
RC2087	(2087, X'827') Unknown remote queue manager.
RC2196	(2196, X'894') Unknown transmission queue.
RC2091	(2091, X'82B') Transmission queue not local.
RC2092	(2092, X'82C') Transmission queue with wrong usage.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

Usage notes

- 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 *OBJDSC* parameter described in “Chapter 32. MQOPEN - Open object” on page 251.

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).
- A process definition, in order to:
 - Inquire about the process attributes (using the MQINQ call).
- The queue manager, in order to:

Usage notes

- 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. If the object being opened is a queue but not a cluster queue, 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 *OPTS* parameter on the corresponding MQOPEN.

If the object being opened is a cluster queue, name resolution can occur at the time of the MQOPEN call, or be deferred until later. The point at which resolution occurs is controlled by the OOBND* options specified on the MQOPEN call:

- OOBND0
- OOBNDN
- OOBNDQ

Refer to the *MQSeries Queue Manager Clusters* book for more information about name resolution for cluster queues.

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. This applies regardless of the open options used, and includes the following:
 - A change to the *BaseQName* attribute of an alias queue that is open.
 - A change to 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.
 - 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.

There is one exception to this, namely 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.
- The *Shareability* local-queue attribute, if there are two or more handles that are currently providing OOINPS 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.
- 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 RC2041; 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.
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 *OBJDSC* parameter of this call (see the *ODON* and *ODMN* fields described in “Chapter 12. MQOD - Object descriptor” on page 137):
 - By specifying for *ODON* the name of a local definition of the remote queue. In this case, *ODMN* refers to the local queue manager, and can be specified as blanks.
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 *ODON* the name of the remote queue as known to the remote queue manager. In this case, *ODMN* 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.

Usage notes

- 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.
- a. Fields in the MQOD structure must be set as follows when opening a distribution list:
 - *ODVER* must be *ODVER2* or greater.
 - *ODOT* must be *OTQ*.
 - *ODON* must be blank or the null string.
 - *ODMN* must be blank or the null string.
 - *ODREC* must be greater than zero.
 - One of *ODORO* and *ODORP* must be zero and the other nonzero.
 - No more than one of *ODRRO* and *ODRRP* can be nonzero.
 - There must be *ODREC* object records, addressed by either *ODORO* or *ODORP*. The object records must be set to the names of the destination queues to be opened.
 - If one of *ODRRO* and *ODRRP* is nonzero, there must be *ODREC* response records present. They are set by the queue manager if the call completes with reason code RC2136.

A version-2 MQOD can also be used to open a single queue that is not in a distribution list, by ensuring that *ODREC* is zero.

- b. Only the following open options are valid in the *OPTS* parameter:
 - OOOUT
 - OOPAS*
 - OOSET*
 - OOALTU
 - OOFIQ
- 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 RC2057. 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 CCOK and RCNONE respectively; if every open fails because none of the queues exists, the parameters are set to CCFAIL and RC2085.
 - 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 CCWARN if at least one open succeeded, and to CCFAIL if all failed.
 - The reason code parameter is set to RC2136.
 - 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 CONONE.

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. If the MQOPEN or MQPUT1 call for a distribution list would cause the number of handles in use by the application to exceed *MaxHandles*, the call fails with reason code RC2017.
 - g. Each destination that is opened successfully has the value of its *OpenOutputCount* attribute incremented by one. If two or more of the destinations in the distribution list actually resolve to the same physical queue, that queue has its *OpenOutputCount* attribute incremented by the number of destinations in the distribution list that resolve to that queue.
 - h. 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.
 - i. It is valid for a distribution list to contain only one destination.
9. An MQOPEN call with the OOBROW 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 GMMUC option.

Multiple browse cursors can be active for a single application by issuing several MQOPEN requests for the same queue.

10. The following notes apply to the use of cluster queues.
- a. When a cluster queue is opened for the first time, and the local queue manager is not a full repository queue manager, the local queue manager obtains information about the cluster queue from a full repository queue manager. When the network is busy, it may take several seconds for the local queue manager to receive the needed information from the repository queue manager. As a result, the application issuing the MQOPEN call may have to wait for up to 10 seconds before control returns from the MQOPEN call. If the local queue manager does not receive the needed information about the cluster queue within this time, the call fails with reason code RC2189.
 - b. When a cluster queue is opened and there are multiple instances of the queue in the cluster, the instance actually opened depends on the options specified on the MQOPEN call:
 - If the options specified include any of the following:
 - OOBRW
 - OOINPQ
 - OOINPX
 - OOINPS
 - OOSET

the instance of the cluster queue opened is required to be the local instance. If there is no local instance of the queue, the MQOPEN call fails.

Usage notes

- If the options specified include none of the above, but do include one or both of the following:

OOINQ
OOOUT

the instance opened is the local instance if there is one, and a remote instance otherwise. The instance chosen by the queue manager can, however, be altered by a cluster workload exit (if there is one).

For more information about cluster queues, refer to the *MQSeries Queue Manager Clusters* book.

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 *OBJDSC* parameter to open the queue. See the description of the MQTMC structure for further details.
12. On OS/400, applications running in compatibility mode are connected automatically to the queue manager by the first MQOPEN call issued by the application (if the application has not already connected to the queue manager by using the MQCONN call).

Applications not running in compatibility mode must issue the MQCONN or MQCONNX call to connect to the queue manager explicitly, before using the MQOPEN call to open an object.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQOPEN(HCONN : OBJDSC : OPTS :
C                               HOBJ : CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
DMQOPEN      PR          EXTPROC('MQOPEN')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object descriptor
D OBJDSC          360A
D* Options that control the action of MQOPEN
D OPTS          10I 0 VALUE
D* Object handle
D HOBJ          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0
```

Chapter 33. MQPUT - Put message

The MQPUT call puts a message on a queue or distribution list. The queue or distribution list must already be open.

Syntax

MQPUT (*HCONN*, *HOBJ*, *MSGDSC*, *PMO*, *BUFLN*, *BUFFER*, *CMPCOD*,
REASON)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

HOBJ (10-digit signed integer) – 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 OOOUT option.

MSGDSC (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 “Chapter 10. MQMD - Message descriptor” on page 83 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 *MDFMT* field in the MQMD must be set to FMMDE to indicate that an MQMDE is present. See “Chapter 11. MQMDE - Message descriptor extension” on page 131 for more details.

PMO (MQPMO) – input/output
Options that control the action of MQPUT.

See “Chapter 14. MQPMO - Put message options” on page 149 for details.

BUFLN (10-digit signed integer) – input
Length of the message in *BUFFER*.

Zero is valid, and indicates that the message contains no application data.

MQPUT — Put message

If the destination is a local queue, or resolves to a local queue, the upper limit for *BUFLen* 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 MFSEGA, 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, *BUFLen* cannot exceed 999 999 999 minus the value of the *MDOFF* field in MQMD. The longest logical message that can be put is therefore 999 999 999 bytes (when *MDOFF* is zero). However, resource constraints imposed by the operating system or environment in which the application is running may result in a lower limit.

If one or both of the above conditions is not satisfied, *BUFLen* 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 LNMHD bytes be subtracted from the *MaxMsgLength* values of the transmission queues when determining the limit for *BUFLen*.

Note: Only failure to comply with condition 1 can be diagnosed synchronously (with reason code RC2030 or RC2031) 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 (1-byte bit string×*BUFLen*) – input
Message data.

This is a buffer containing the application data to be sent.

If *BUFFER* contains character and/or numeric data, the *MDCSI* and *MDENC* fields in the *MSGDSC* 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 ENNAT, respectively).

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is **CCOK**:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is **CCWARN**:

RC2136

(2136, X'858') Multiple reason codes returned.

RC2049

(2049, X'801') Message Priority exceeds maximum value supported.

RC2104

(2104, X'838') Report option(s) in message descriptor not recognized.

If *CMPCOD* is **CCFAIL**:

RC2004

(2004, X'7D4') Buffer parameter not valid.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2266

(2266, X'8DA') Cluster workload exit failed.

RC2189

(2189, X'88D') Cluster name resolution failed.

RC2269

(2269, X'8DD') Cluster resource error.

RC2009

(2009, X'7D9') Connection to queue manager lost.

RC2097

(2097, X'831') Queue handle referred to does not save context.

RC2098

(2098, X'832') Context not available for queue handle referred to.

RC2135

(2135, X'857') Distribution header structure not valid.

RC2013

(2013, X'7DD') Expiry time not valid.

RC2014

(2014, X'7DE') Feedback code not valid.

RC2258

(2258, X'8D2') Group identifier not valid.

RC2018

(2018, X'7E2') Connection handle not valid.

MQPUT — Put message

- RC2019**
(2019, X'7E3') Object handle not valid.
- RC2241**
(2241, X'8C1') Message group not complete.
- RC2242**
(2242, X'8C2') Logical message not complete.
- RC2185**
(2185, X'889') Inconsistent persistence specification.
- RC2245**
(2245, X'8C5') Inconsistent unit-of-work specification.
- RC2026**
(2026, X'7EA') Message descriptor not valid.
- RC2248**
(2248, X'8C8') Message descriptor extension not valid.
- RC2027**
(2027, X'7EB') Missing reply-to queue.
- RC2249**
(2249, X'8C9') Message flags not valid.
- RC2250**
(2250, X'8CA') Message sequence number not valid.
- RC2030**
(2030, X'7EE') Message length greater than maximum for queue.
- RC2031**
(2031, X'7EF') Message length greater than maximum for queue manager.
- RC2029**
(2029, X'7ED') Message type in message descriptor not valid.
- RC2136**
(2136, X'858') Multiple reason codes returned.
- RC2270**
(2270, X'8DE') No destination queues available.
- RC2039**
(2039, X'7F7') Queue not open for output.
- RC2093**
(2093, X'82D') Queue not open for pass all context.
- RC2094**
(2094, X'82E') Queue not open for pass identity context.
- RC2095**
(2095, X'82F') Queue not open for set all context.
- RC2096**
(2096, X'830') Queue not open for set identity context.
- RC2041**
(2041, X'7F9') Object definition changed since opened.
- RC2101**
(2101, X'835') Object damaged.
- RC2251**
(2251, X'8CB') Message segment offset not valid.
- RC2137**
(2137, X'859') Object not opened successfully.
- RC2046**
(2046, X'7FE') Options not valid or not consistent.
- RC2252**
(2252, X'8CC') Original length not valid.
- RC2149**
(2149, X'865') PCF structures not valid.

- RC2047**
(2047, X'7FF') Persistence not valid.
- RC2048**
(2048, X'800') Message on a temporary dynamic queue cannot be persistent.
- RC2173**
(2173, X'87D') Put-message options structure not valid.
- RC2158**
(2158, X'86E') Put message record flags not valid.
- RC2050**
(2050, X'802') Message priority not valid.
- RC2051**
(2051, X'803') Put calls inhibited for the queue.
- RC2159**
(2159, X'86F') Put message records not valid.
- RC2052**
(2052, X'804') Queue has been deleted.
- RC2053**
(2053, X'805') Queue already contains maximum number of messages.
- RC2058**
(2058, X'80A') Queue manager name not valid or not known.
- RC2059**
(2059, X'80B') Queue manager not available for connection.
- RC2161**
(2161, X'871') Queue manager quiescing.
- RC2162**
(2162, X'872') Queue manager shutting down.
- RC2056**
(2056, X'808') No space available on disk for queue.
- RC2154**
(2154, X'86A') Number of records present not valid.
- RC2061**
(2061, X'80D') Report options in message descriptor not valid.
- RC2156**
(2156, X'86C') Response records not valid.
- RC2102**
(2102, X'836') Insufficient system resources available.
- RC2253**
(2253, X'8CD') Length of data in message segment is zero.
- RC2188**
(2188, X'88C') Call rejected by cluster workload exit.
- RC2071**
(2071, X'817') Insufficient storage available.
- RC2024**
(2024, X'7E8') No more messages can be handled within current unit of work.
- RC2072**
(2072, X'818') Syncpoint support not available.
- RC2195**
(2195, X'893') Unexpected error occurred.
- RC2255**
(2255, X'8CF') Unit of work not available for the queue manager to use.
- RC2257**
(2257, X'8D1') Wrong version of MQMD supplied.

MQPUT — Put message

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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 OOOOUT 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. If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved provided that the conditions detailed below are satisfied. Some conditions apply to both local and remote destination queues; other conditions apply only to remote destination queues.

Conditions for local and remote destination queues

- All of the MQPUT calls are within the same unit of work, or none of them is within a unit of work.

Be aware that 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.
- 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 OS/400, the application is the job.
- The messages all have the same priority.

Additional conditions for remote destination queues

- There is only one path from the sending 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 queue manager cannot be guaranteed.
- Messages are not placed temporarily on dead-letter queues at the sending, intermediate, or destination queue managers.

If one or more of the messages is put temporarily on a dead-letter queue (for example, because a transmission queue or the destination queue is temporarily full), the messages can arrive on the destination queue out of sequence.
- The messages are either all persistent or all nonpersistent.

If a channel on the route between the sending and destination queue managers has its *CDNPM* attribute set to NPFAST, nonpersistent messages can

jump ahead of persistent messages, resulting in the order of persistent messages relative to nonpersistent messages not being preserved. However, the order of persistent messages relative to each other, and of nonpersistent messages relative to each other, is preserved.

If these conditions are not satisfied, message groups can be used to preserve message order, but note that this requires both the sending and receiving applications to use the message-grouping support. For more information about message groups, see:

- *MDMFL* field in MQMD
- *PMLOGO* option in MQPMO
- *GMLOGO* option in MQGMO

3. The following notes apply to the use of distribution lists.

- 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 *PMREC* 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 *PMVER* field must be set to *PMVER2*.

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 *PMREC* 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 *CCOK* and *RCNONE* respectively; if every put fails because all of the queues are inhibited for puts, the parameters are set to *CCFAIL* and *RC2051*.

- 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 *CCWARN* if at least one put succeeded, and to *CCFAIL* if all failed.
 - The reason code parameter is set to *RC2136*.
 - 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 *CCFAIL* and *RC2137*; that destination is included in *PMIDC*.

- 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

Usage notes

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 “Chapter 38. Attributes for local queues and model queues” on page 299).

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 PRQDEF or PEQDEF), 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.
4. 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 *BUFLN* parameter on the MQPUT or MQPUT1 call. A PCF message is a message that has one of the following format names:
 - FMADMN
 - FMEVNT
 - FMPCF
 - 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.)
5. The *BUFFER* parameter shown in the RPG programming example is declared as a string; this restricts the maximum length of the parameter to 256 bytes. If a larger buffer is required, the parameter should be declared instead as a structure, or as a field in a physical file. This will increase the maximum length possible to approximately 32 KB.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQPUT(HCONN : HOBJ : MSGDSC : PMO :
C                               BUFLN : BUFFER : CMPCOD :
C                               REASON)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
DMQPUT          PR          EXTPROC('MQPUT')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object handle
D HOBJ          10I 0 VALUE
D* Message descriptor
D MSGDSC          364A
D* Options that control the action of MQPUT
D PMO          176A
D* Length of the message in BUFFER
D BUFLN          10I 0 VALUE
D* Message data
D BUFFER          * VALUE
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0
```

Usage notes

Chapter 34. MQPUT1 - Put one message

The MQPUT1 call puts one message on a queue. The queue need not be open.

Syntax

MQPUT1 (*HCONN*, *OBJDSC*, *MSGDSC*, *PMO*, *BUFLN*, *BUFFER*, *CMPCOD*,
REASON)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH

Default connection handle.

OBJDSC (MQOD) – input/output
Object descriptor.

This is a structure which identifies the queue to which the message is added. See “Chapter 12. MQOD - Object descriptor” on page 137 for details.

The application must be authorized to open the queue for output. The queue must **not** be a model queue.

MSGDSC (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 “Chapter 10. MQMD - Message descriptor” on page 83 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 *MDFMT* field in the MQMD must be set to FMMDE to indicate that an MQMDE is present. See “Chapter 11. MQMDE - Message descriptor extension” on page 131 for more details.

PMO (MQPMO) – input/output
Options that control the action of MQPUT1.

See “Chapter 14. MQPMO - Put message options” on page 149 for details.

BUFLN (10-digit signed integer) – input
Length of the message in *BUFFER*.

MQPUT1 — Put one message

Zero is valid, and indicates that the message contains no application data. The upper limit depends on various factors; see the description of the *BUFLen* parameter of the MQPUT call for further details.

BUFFER (1-byte bit string×*BUFLen*) – input
Message data.

This is a buffer containing the application message data to be sent.

If *BUFFER* contains character and/or numeric data, the *MDCSI* and *MDENC* fields in the *MSGDSC* 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 ENNAT, respectively).

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK

Successful completion.

CCWARN

Warning (partial completion).

CCFAIL

Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is CCOK:

RCNONE

(0, X'000') No reason to report.

If *CMPCOD* is CCWARN:

RC2136

(2136, X'858') Multiple reason codes returned.

RC2241

(2241, X'8C1') Message group not complete.

RC2242

(2242, X'8C2') Logical message not complete.

RC2049

(2049, X'801') Message Priority exceeds maximum value supported.

RC2104

(2104, X'838') Report option(s) in message descriptor not recognized.

If *CMPCOD* is CCFAIL:

RC2001

(2001, X'7D1') Alias base queue not a valid type.

RC2004

(2004, X'7D4') Buffer parameter not valid.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

RC2219

(2219, X'8AB') MQI call reentered before previous call complete.

RC2266	(2266, X'8DA') Cluster workload exit failed.
RC2189	(2189, X'88D') Cluster name resolution failed.
RC2269	(2269, X'8DD') Cluster resource error.
RC2009	(2009, X'7D9') Connection to queue manager lost.
RC2097	(2097, X'831') Queue handle referred to does not save context.
RC2098	(2098, X'832') Context not available for queue handle referred to.
RC2198	(2198, X'896') Default transmission queue not local.
RC2199	(2199, X'897') Default transmission queue usage error.
RC2135	(2135, X'857') Distribution header structure not valid.
RC2013	(2013, X'7DD') Expiry time not valid.
RC2014	(2014, X'7DE') Feedback code not valid.
RC2258	(2258, X'8D2') Group identifier not valid.
RC2017	(2017, X'7E1') No more handles available.
RC2018	(2018, X'7E2') Connection handle not valid.
RC2026	(2026, X'7EA') Message descriptor not valid.
RC2248	(2248, X'8C8') Message descriptor extension not valid.
RC2027	(2027, X'7EB') Missing reply-to queue.
RC2249	(2249, X'8C9') Message flags not valid.
RC2250	(2250, X'8CA') Message sequence number not valid.
RC2030	(2030, X'7EE') Message length greater than maximum for queue.
RC2031	(2031, X'7EF') Message length greater than maximum for queue manager.
RC2029	(2029, X'7ED') Message type in message descriptor not valid.
RC2136	(2136, X'858') Multiple reason codes returned.
RC2270	(2270, X'8DE') No destination queues available.
RC2035	(2035, X'7F3') Not authorized for access.
RC2101	(2101, X'835') Object damaged.
RC2042	(2042, X'7FA') Object already open with conflicting options.

MQPUT1 — Put one message

- RC2155**
(2155, X'86B') Object records not valid.
- RC2043**
(2043, X'7FB') Object type not valid.
- RC2044**
(2044, X'7FC') Object descriptor structure not valid.
- RC2251**
(2251, X'8CB') Message segment offset not valid.
- RC2046**
(2046, X'7FE') Options not valid or not consistent.
- RC2252**
(2252, X'8CC') Original length not valid.
- RC2149**
(2149, X'865') PCF structures not valid.
- RC2047**
(2047, X'7FF') Persistence not valid.
- RC2048**
(2048, X'800') Message on a temporary dynamic queue cannot be persistent.
- RC2173**
(2173, X'87D') Put-message options structure not valid.
- RC2158**
(2158, X'86E') Put message record flags not valid.
- RC2050**
(2050, X'802') Message priority not valid.
- RC2051**
(2051, X'803') Put calls inhibited for the queue.
- RC2159**
(2159, X'86F') Put message records not valid.
- RC2052**
(2052, X'804') Queue has been deleted.
- RC2053**
(2053, X'805') Queue already contains maximum number of messages.
- RC2058**
(2058, X'80A') Queue manager name not valid or not known.
- RC2059**
(2059, X'80B') Queue manager not available for connection.
- RC2161**
(2161, X'871') Queue manager quiescing.
- RC2162**
(2162, X'872') Queue manager shutting down.
- RC2056**
(2056, X'808') No space available on disk for queue.
- RC2057**
(2057, X'809') Queue type not valid.
- RC2154**
(2154, X'86A') Number of records present not valid.
- RC2184**
(2184, X'888') Remote queue name not valid.
- RC2061**
(2061, X'80D') Report options in message descriptor not valid.
- RC2102**
(2102, X'836') Insufficient system resources available.
- RC2156**
(2156, X'86C') Response records not valid.

RC2063	(2063, X'80F') Security error occurred.
RC2253	(2253, X'8CD') Length of data in message segment is zero.
RC2188	(2188, X'88C') Call rejected by cluster workload exit.
RC2071	(2071, X'817') Insufficient storage available.
RC2024	(2024, X'7E8') No more messages can be handled within current unit of work.
RC2072	(2072, X'818') Syncpoint support not available.
RC2195	(2195, X'893') Unexpected error occurred.
RC2082	(2082, X'822') Unknown alias base queue.
RC2197	(2197, X'895') Unknown default transmission queue.
RC2085	(2085, X'825') Unknown object name.
RC2086	(2086, X'826') Unknown object queue manager.
RC2087	(2087, X'827') Unknown remote queue manager.
RC2196	(2196, X'894') Unknown transmission queue.
RC2255	(2255, X'8CF') Unit of work not available for the queue manager to use.
RC2257	(2257, X'8D1') Wrong version of MQMD supplied.
RC2091	(2091, X'82B') Transmission queue not local.
RC2092	(2092, X'82C') Transmission queue with wrong usage.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

Usage notes

- 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 OOOUT 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.
- If an application puts a sequence of messages on the same queue without using message groups, the order of those messages is preserved provided that certain

Usage notes

conditions are satisfied. However, in most environments the MQPUT1 call does not satisfy these conditions, and so does not preserve message order. The MQPUT call must be used instead in these environments. See the usage notes in the description of the MQPUT call for details.

3. The MQPUT1 call can be used to put messages to distribution lists. For general information about this, see usage note 8 on page 262 for the MQOPEN call, and usage note 3 on page 271 for the MQPUT call.

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 RC2137 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 CCWARN, 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 RC2136.

4. If the MQPUT1 call is used to put a message on a cluster queue, the call behaves as though OOBNDN had been specified on the MQOPEN call.
5. 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 4 on page 272 for the MQPUT call.
6. If more than one of the warning situations arise (see the *CMPCOD* parameter), the reason code returned is the *first* one in the following list that applies:
 - a. RC2136
 - b. RC2242
 - c. RC2241
 - d. RC2049 or RC2104
7. The *BUFFER* parameter shown in the RPG programming example is declared as a string; this restricts the maximum length of the parameter to 256 bytes. If a larger buffer is required, the parameter should be declared instead as a structure, or as a field in a physical file. This will increase the maximum length possible to approximately 32 KB.

RPG invocation

```

C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQPUT1(HCONN : OBJDSC : MSGDSC :
C                               PMO : BUFLN : BUFFER :
C                               CMPCOD : REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQPUT1      PR          EXTPROC('MQPUT1')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object descriptor
D OBJDSC          360A
D* Message descriptor
D MSGDSC          364A
D* Options that control the action of MQPUT1
D PMO            176A
D* Length of the message in BUFFER
D BUFLN          10I 0 VALUE
D* Message data
D BUFFER          *   VALUE
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0

```

Usage notes

Chapter 35. 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.

Syntax

MQSET (*HCONN*, *HOBJ*, *SELCNT*, *SELS*, *IACNT*, *INTATR*, *CALEN*,
CHRATR, *CMPCOD*, *REASON*)

Parameters

HCONN (10-digit signed integer) – 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 for applications running in compatibility mode, the MQCONN call can be omitted, and the following value specified for *HCONN*:

HCDEFH
Default connection handle.

HOBJ (10-digit signed integer) – 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 OOSSET option.

SELCNT (10-digit signed integer) – input
Count of selectors.

This is the count of selectors that are supplied in the *SELS* array. It is the number of attributes that are to be set. Zero is a valid value. The maximum number allowed is 256.

SELS (10-digit signed integer×*SELCNT*) – input
Array of attribute selectors.

This is an array of *SELCNT* 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 IA* and CA* values are allowed; these values are listed below.

Selectors can be specified in any order. Attribute values that correspond to integer attribute selectors (IA* selectors) must be specified in *INTATR* in the same order in which these selectors occur in *SELS*. Attribute values that correspond to character attribute selectors (CA* selectors) must be specified in *CHRATR* in the same order in which those selectors occur. IA* selectors can be interleaved with the CA* selectors; only the relative order within each type is important.

MQSET — Set object attributes

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 IA* selectors reside within the range IAFRST through IALAST, and the CA* selectors within the range CAFRST through CALAST. For each range, the constants IALSTU and CALSTU define the highest value that the queue manager will accept.
2. If all the IA* selectors occur first, the same element numbers can be used to address corresponding elements in the *SELS* and *INTATR* arrays.

For the CA* selectors in the following descriptions, the constant that defines the length in bytes of the string that is required in *CHRATR* is given in parentheses.

Selectors for all types of queue

IAIPUT

Whether put operations are allowed.

Selectors for local queues

CATRGD

Trigger data (LNTRGD).

IADIST

Distribution list support.

IAIGET

Whether get operations are allowed.

IATRGC

Trigger control.

IATRGD

Trigger depth.

IATRGP

Threshold message priority for triggers.

IATRGT

Trigger type.

Selectors for alias queues

IAIGET

Whether get operations are allowed.

No other attributes can be set using this call.

IACNT (10-digit signed integer) – input
Count of integer attributes.

This is the number of elements in the *INTATR* array, and must be at least the number of IA* selectors in the *SELS* parameter. Zero is a valid value if there are none.

INTATR (10-digit signed integer×*IACNT*) – input
Array of integer attributes.

This is an array of *IACNT* integer attribute values. These attribute values must be in the same order as the IA* selectors in the *SELS* array.

CALEN (10-digit signed integer) – input
Length of character attributes buffer.

MQSET — Set object attributes

This is the length in bytes of the *CHRATR* parameter, and must be at least the sum of the lengths of the character attributes specified in the *SELS* array. Zero is a valid value if there are no *CA** selectors in *SELS*.

CHRATR (1-byte character string×*CALEN*) – input
Character attributes.

This is the buffer containing the character attribute values, concatenated together. The length of the buffer is given by the *CALEN* parameter.

The characters attributes must be specified in the same order as the *CA** selectors in the *SELS* array. The length of each character attribute is fixed (see *SELS*). If the value to be set for an attribute contains fewer nonblank characters than the defined length of the attribute, the value in *CHRATR* must be padded to the right with blanks to make the attribute value match the defined length of the attribute.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK
Successful completion.
CCFAIL
Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is **CCOK**:
RCNONE
(0, X'000') No reason to report.

If *CMPCOD* is **CCFAIL**:
RC2219
(2219, X'8AB') MQI call reentered before previous call complete.
RC2006
(2006, X'7D6') Length of character attributes not valid.
RC2007
(2007, X'7D7') Character attributes string not valid.
RC2009
(2009, X'7D9') Connection to queue manager lost.
RC2018
(2018, X'7E2') Connection handle not valid.
RC2019
(2019, X'7E3') Object handle not valid.
RC2020
(2020, X'7E4') Value for inhibit-get or inhibit-put queue attribute not valid.
RC2021
(2021, X'7E5') Count of integer attributes not valid.
RC2023
(2023, X'7E7') Integer attributes array not valid.
RC2040
(2040, X'7F8') Queue not open for set.
RC2041
(2041, X'7F9') Object definition changed since opened.

MQSET — Set object attributes

RC2101	(2101, X'835') Object damaged.
RC2052	(2052, X'804') Queue has been deleted.
RC2058	(2058, X'80A') Queue manager name not valid or not known.
RC2059	(2059, X'80B') Queue manager not available for connection.
RC2162	(2162, X'872') Queue manager shutting down.
RC2102	(2102, X'836') Insufficient system resources available.
RC2065	(2065, X'811') Count of selectors not valid.
RC2067	(2067, X'813') Attribute selector not valid.
RC2066	(2066, X'812') Count of selectors too big.
RC2071	(2071, X'817') Insufficient storage available.
RC2075	(2075, X'81B') Value for trigger-control attribute not valid.
RC2076	(2076, X'81C') Value for trigger-depth attribute not valid.
RC2077	(2077, X'81D') Value for trigger-message-priority attribute not valid.
RC2078	(2078, X'81E') Value for trigger-type attribute not valid.
RC2195	(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see “Appendix A. Return codes” on page 357.

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 “Chapter 31. MQINQ - Inquire about object attributes” on page 241 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 OOSSET option, you can use the MQSET call to set the attributes of the dynamic queue that is created by the MQOPEN call.

5. If the object being set is a cluster queue, there must be a local instance of the cluster queue for the open to succeed.
6. For more information about object attributes, see Chapter 36. Attributes of MQSeries objects.

RPG invocation

```

C*..1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQSET(HCONN : HOBJ : SELCNT :
C                      SELS(1) : IACNT : INTATR(1) :
C                      CALEN : CHRATR : CMPCOD :
C                      REASON)

```

The prototype definition for the call is:

```

D*..1.....2.....3.....4.....5.....6.....7..
DMQSET      PR          EXTPROC('MQSET')
D* Connection handle
D HCONN          10I 0 VALUE
D* Object handle
D HOBJ          10I 0 VALUE
D* Count of selectors
D SELCNT        10I 0 VALUE
D* Array of attribute selectors
D SELS          10I 0
D* Count of integer attributes
D IACNT         10I 0 VALUE
D* Array of integer attributes
D INTATR        10I 0
D* Length of character attributes buffer
D CALEN         10I 0 VALUE
D* Character attributes
D CHRATR        * VALUE
D* Completion code
D CMPCOD        10I 0
D* Reason code qualifying CMPCOD
D REASON        10I 0

```

RPG invocation

Part 3. Attributes of objects

Chapter 36. Attributes of MQSeries objects	291
Chapter 37. Attributes for all queues.	293
Chapter 38. Attributes for local queues and model queues.	299
Chapter 39. Attributes for local definitions of remote queues	313
Chapter 40. Attributes for alias queues	315
Chapter 41. Attributes for namelists	317
Chapter 42. Attributes for process definitions	319
Chapter 43. Attributes for the queue manager	323

Object attributes

Chapter 36. Attributes of MQSeries objects

MQSeries objects consist of:

- Channels
- Queues
- Queue managers
- Namelists
- Processes

This part of the book describes the attributes (or properties) of MQSeries objects that are accessible through the API, which are queues, queue managers, namelists, and processes.

Namelists are supported in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, and Windows NT.

Process definitions are not supported in the following environments: 16-bit Windows, 32-bit Windows, and VSE/ESA.

The attributes are grouped according to the type of object to which they apply; see:

- “Chapter 37. Attributes for all queues” on page 293
- “Chapter 38. Attributes for local queues and model queues” on page 299
- “Chapter 39. Attributes for local definitions of remote queues” on page 313
- “Chapter 40. Attributes for alias queues” on page 315
- “Chapter 41. Attributes for namelists” on page 317
- “Chapter 42. Attributes for process definitions” on page 319
- “Chapter 43. Attributes for the queue manager” on page 323

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

Chapter 37. 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 49. Attributes for all queues

Attribute	Description	Page
<i>AlterationDate</i>	Date when definition was last changed	293
<i>AlterationTime</i>	Time when definition was last changed	293
<i>ClusterName</i>	Name of cluster to which queue belongs	293
<i>ClusterNameList</i>	Name of namelist object containing names of clusters to which queue belongs	294
<i>DefBind</i>	Default binding	294
<i>DefPersistence</i>	Default message persistence	294
<i>DefPriority</i>	Default message priority	295
<i>InhibitGet</i>	Controls whether get operations for the queue are allowed	295
<i>InhibitPut</i>	Controls whether put operations for the queue are allowed	296
<i>QDesc</i>	Queue description	296
<i>QName</i>	Queue name	297
<i>QType</i>	Queue type	297
<i>Scope</i>	Controls whether an entry for the queue also exists in a cell directory	297

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

ClusterName (48-byte character string)

Name of cluster to which queue belongs.

This is the name of the cluster to which the queue belongs. If the queue belongs to more than one cluster, *ClusterNameList* specifies the name of a

Attributes - all queues

namelist object that identifies the clusters, and *ClusterName* is blank. At least one of *ClusterName* and *ClusterNameList* must be blank. This attribute does not apply to model queues.

To determine the value of this attribute, use the *CACLN* selector with the *MQINQ* call. The length of this attribute is given by *LNCLUN*.

ClusterNameList (48-byte character string)

Name of namelist object containing names of clusters to which queue belongs.

This is the name of a namelist object that contains the names of clusters to which this queue belongs. If the queue belongs to only one cluster, the namelist object contains only one name. Alternatively, *ClusterName* can be used to specify the name of the cluster, in which case *ClusterNameList* is blank. At least one of *ClusterName* and *ClusterNameList* must be blank. This attribute does not apply to model queues.

To determine the value of this attribute, use the *CACLNL* selector with the *MQINQ* call. The length of this attribute is given by *LNNLN*.

DefBind (10-digit signed integer)

Default binding.

This is the default binding that is used when *OOBNDQ* is specified on the *MQOPEN* call and the queue is a cluster queue. This attribute does not apply to model queues.

The value is one of the following:

BNDOPN

Binding fixed by *MQOPEN* call.

BNDNOT

Binding not fixed.

To determine the value of this attribute, use the *IADBND* selector with the *MQINQ* call.

DefPersistence (10-digit signed integer)

Default message persistence.

This is the default persistence of messages on the queue. It applies if *PEQDEF* 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 *MQPUT* or *MQPUT1* call. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

The value is one of the following:

PEPER

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 placed on temporary dynamic queues; persistent messages can however be placed on permanent dynamic queues, and predefined queues.

PENPER

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 IADPER selector with the MQINQ call.

DefPriority (10-digit signed integer)

Default message priority

This is the default priority for messages on the queue. This applies if PRQDEF 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. This could be:

- An alias queue
- A local queue
- A local definition of a remote queue
- A queue-manager alias
- A transmission queue (for example, the *DefXmitQName* queue)

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 MSPRIO, the logical position at which a message is placed on the queue is dependent on the value of the *MDPRI* field in the message descriptor.
- If the *MsgDeliverySequence* attribute is MSFIFO, 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 *MDPRI* field in the message descriptor. However, the *MDPRI* field retains the value specified by the application that put the message. See the *MsgDeliverySequence* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299 for more information.

Priorities are in the range zero (lowest) through *MaxPriority* (highest); see the *MaxPriority* attribute described in “Chapter 43. Attributes for the queue manager” on page 323.

To determine the value of this attribute, use the IADPRI selector with the MQINQ call.

InhibitGet (10-digit signed integer)

Controls whether get operations for this queue are allowed.

This attribute applies only to local, model, and alias queues.

Attributes - all 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:

QAGETI

Get operations are inhibited.

MQGET calls fail with reason code RC2016. This includes MQGET calls that specify GMBRWF or GMBRWN.

Note: If an MQGET call operating within a unit of work completes successfully, changing the value of the *InhibitGet* attribute subsequently to QAGETI does not prevent the unit of work being committed.

QAGETA

Get operations are allowed.

To determine the value of this attribute, use the IAIGET selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

InhibitPut (10-digit signed integer)

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:

QAPUTI

Put operations are inhibited.

MQPUT and MQPUT1 calls fail with reason code RC2051.

Note: If an MQPUT call operating within a unit of work completes successfully, changing the value of the *InhibitPut* attribute subsequently to QAPUTI does not prevent the unit of work being committed.

QAPUTA

Put operations are allowed.

To determine the value of this attribute, use the IAIPUT selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

QDesc (64-byte character string)

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 CAQD selector with the MQINQ call. The length of this attribute is given by LNQD.

QName (48-byte character string)
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 QTLOC queue and a QTALS queue cannot have the same name.

To determine the value of this attribute, use the CAQN selector with the MQINQ call. The length of this attribute is given by LNQN.

QType (10-digit signed integer)
Queue type.

This attribute has one of the following values:

QTALS

Alias queue definition.

QTCLUS

Cluster queue.

QTLOC

Local queue.

QTREM

Local definition of a remote queue.

To determine the value of this attribute, use the IAQTYP selector with the MQINQ call.

Scope (10-digit signed integer)
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:

SCOQM

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.

SCOCEL

Cell scope.

Attributes - all queues

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 queue 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 IASCOP selector with the MQINQ call.

Support for this attribute is subject to the following restrictions:

- On OS/400, the attribute is supported, but only SCOQM is valid.

Chapter 38. 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 50. Attributes for local and model queues

Attribute	Description	Page
<i>BackoutRequeueQName</i>	Excessive backout requeue queue name	300
<i>BackoutThreshold</i>	Backout threshold	300
<i>CreationDate</i>	Date the queue was created	300
<i>CreationTime</i>	Time the queue was created	300
<i>CurrentQDepth</i>	Current queue depth	300
<i>DefinitionType</i>	Queue definition type	301
<i>DefInputOpenOption</i>	Default input open option	302
<i>DistLists</i>	Distribution list support	302
<i>HardenGetBackout</i>	Whether to maintain an accurate backout count	303
<i>InitiationQName</i>	Name of initiation queue	304
<i>MaxMsgLength</i>	Maximum message length in bytes	304
<i>MaxQDepth</i>	Maximum queue depth	304
<i>MsgDeliverySequence</i>	Message delivery sequence	305
<i>OpenInputCount</i>	Number of opens for input	306
<i>OpenOutputCount</i>	Number of opens for output	306
<i>ProcessName</i>	Process name	306
<i>QDepthHighEvent</i>	Controls whether Queue Depth High events are generated	307
<i>QDepthHighLimit</i>	High limit for queue depth	307
<i>QDepthLowEvent</i>	Controls whether Queue Depth Low events are generated	307
<i>QDepthLowLimit</i>	Low limit for queue depth	308
<i>QDepthMaxEvent</i>	Controls whether Queue Full events are generated	308
<i>QServiceInterval</i>	Target for queue service interval	308
<i>QServiceIntervalEvent</i>	Controls whether Service Interval High or Service Interval OK events are generated	308
<i>RetentionInterval</i>	Retention interval	309
<i>Shareability</i>	Queue shareability	310
<i>TriggerControl</i>	Trigger control	310
<i>TriggerData</i>	Trigger data	310
<i>TriggerDepth</i>	Trigger depth	311
<i>TriggerMsgPriority</i>	Threshold message priority for triggers	311
<i>TriggerType</i>	Trigger type	311
<i>Usage</i>	Queue usage	312

Attributes - local and model queues

BackoutRequeueQName (48-byte character string)
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 CABRQN selector with the MQINQ call. The length of this attribute is given by LNQN.

BackoutThreshold (10-digit signed integer)
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 IABTHR selector with the MQINQ call.

CreationDate (12-byte character string)
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 CACRTD selector with the MQINQ call. The length of this attribute is given by LNCRTD.

CreationTime (8-byte character string)
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 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 CACRTT selector with the MQINQ call. The length of this attribute is given by LNCRTT.

CurrentQDepth (10-digit signed integer)
Current queue depth.

Attributes - local and model queues

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 *MDEXP* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

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 IACDEP selector with the MQINQ call.

DefinitionType (10-digit signed integer)
Queue definition type.

This indicates how the queue was defined. It is one of the following:

QDPRE

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 “Chapter 43. Attributes for the queue manager” on page 323).

QDPERM

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 QDPERM for the *DefinitionType* attribute. This type of queue can be deleted using the MQCLOSE call. See “Chapter 25. MQCLOSE - Close object” on page 211 for more details.

QDTEMP

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

Attributes - local and model queues

QDTEMP 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 IADEFT selector with the MQINQ call.

DefInputOpenOption (10-digit signed integer)

Default input open option.

This is the default way in which the queue should be opened for input. It applies if the OOINPQ option is specified on the MQOPEN call when the queue is opened. It is one of the following:

OOINPX

Open queue to get messages with exclusive access.

The queue is opened for use with subsequent MQGET calls. The call fails with reason code RC2042 if the queue is currently open by this or another application for input of any type (OOINPS or OOINPX).

OOINPS

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 OOINPS, but fails with reason code RC2042 if the queue is currently open with OOINPX.

To determine the value of this attribute, use the IADINP selector with the MQINQ call.

DistLists (10-digit signed integer)

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:

DLSUPP

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.

DLNSUP

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 IADIST selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

HardenGetBackout (10-digit signed integer)

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 *MDBOC* 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 QABH only if it is essential that the count is accurate.

On OS/400, the message backout count is always hardened, regardless of the setting of this attribute.

The following values are possible:

QABH

Backout count remembered.

Hardening is used to ensure that the backout count for messages on this queue is accurate.

QABNH

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.

Attributes - local and model queues

To determine the value of this attribute, use the IAHGB selector with the MQINQ call.

InitiationQName (48-byte character string)
Name of initiation queue.

This is the name of a queue defined on the local queue manager; the queue must be of type QTLOC. 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 CAINIQ selector with the MQINQ call. The length of this attribute is given by LNQN.

MaxMsgLength (10-digit signed integer)
Maximum message length in bytes.

This is an upper limit for the length of the longest *physical* message that can be placed on the queue. However, because the *MaxMsgLength* local-queue attribute can be set independently of the *MaxMsgLength* queue-manager attribute, the actual upper limit for the length of the longest physical message that can be placed on the queue is the lesser of those two values.

If the queue manager supports segmentation, it is possible for an application to put a *logical* message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MFSEGA flag in MQMD. If that flag is specified, the upper limit for the length of a logical message is 999 999 999 bytes, but usually resource constraints imposed by the operating system, or by the environment in which the application is running, will result in a lower limit.

An attempt to place on the queue a message that is too long fails with reason code:

- RC2030 if the message is too big for the queue
- RC2031 if the message is too big for the queue manager, but not too big for the queue

The lower limit for the *MaxMsgLength* attribute is zero. The upper limit is determined by the environment:

- On OS/400, the maximum message length is 100 MB (104 857 600 bytes).

For more information, see the *BUFLN* parameter described in “Chapter 33. MQPUT - Put message” on page 265.

To determine the value of this attribute, use the IAMLEN selector with the MQINQ call.

MaxQDepth (10-digit signed integer)
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 RC2053.

Attributes - local and model queues

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 OS/400, 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 IAMDEP selector with the MQINQ call.

MsgDeliverySequence (10-digit signed integer)
Message delivery sequence.

This determines the order in which messages are returned to the application by the MQGET call:

MSFIFO

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 the priority of the message.

MSPRIO

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).

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 MSFIFO when the message arrives, the message is placed on the queue as though its priority were *DefPriority*. This does not affect the value of the *MDPRI* field in the message descriptor of the message; that field retains the value it had when the message was first put.
- If *MsgDeliverySequence* is MSPRIO when the message arrives, the message is placed on the queue at the place appropriate to the priority given by the *MDPRI* field in the message descriptor.

If the value of the *MsgDeliverySequence* attribute is changed while there are messages on the queue, the order of the messages on the queue is not changed.

If the value of the *DefPriority* attribute is changed while there are messages on the queue, the messages will not necessarily be delivered in

Attributes - local and model queues

FIFO order, even though the *MsgDeliverySequence* attribute is set to MSFIFO; those that were placed on the queue at the higher priority are delivered first.

To determine the value of this attribute, use the IAMDS selector with the MQINQ call.

OpenInputCount (10-digit signed integer)

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 IAOIC selector with the MQINQ call.

OpenOutputCount (10-digit signed integer)

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 IAIOC selector with the MQINQ call.

ProcessName (48-byte character string)

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.

Attributes - local and model queues

To determine the value of this attribute, use the CAPRON selector with the MQINQ call. The length of this attribute is given by LNPRON.

QDepthHighEvent (10-digit signed integer)

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:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IAQDHE selector with the MQINQ call.

QDepthHighLimit (10-digit signed integer)

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 IAQDHL selector with the MQINQ call.

QDepthLowEvent (10-digit signed integer)

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:

EVRDIS

Event reporting disabled.

Attributes - local and model queues

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IAQDLE selector with the MQINQ call.

QDepthLowLimit (10-digit signed integer)

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 IAQDLL selector with the MQINQ call.

QDepthMaxEvent (10-digit signed integer)

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:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IAQDME selector with the MQINQ call.

QServiceInterval (10-digit signed integer)

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 IAQSI selector with the MQINQ call.

QServiceIntervalEvent (10-digit signed integer)

Controls whether Service Interval High or Service Interval OK events are generated.

Attributes - local and model queues

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:

QSIEHI

Queue Service Interval High events enabled.

- Queue Service Interval High events are **enabled** and
- Queue Service Interval OK events are **disabled**.

QSIEOK

Queue Service Interval OK events enabled.

- Queue Service Interval High events are **disabled** and
- Queue Service Interval OK events are **enabled**.

QSIENO

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 IAQSIE selector with the MQINQ call.

RetentionInterval (10-digit signed integer)
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 IARINT selector with the MQINQ call.

Attributes - local and model queues

Shareability (10-digit signed integer)

Queue shareability.

This indicates whether the queue can be opened for input multiple times concurrently. It is one of the following:

QASHR

Queue is shareable.

Multiple opens with the OOINPS option are allowed.

QANSHR

Queue is not shareable.

An MQOPEN call with the OOINPS option is treated as OOINPX.

To determine the value of this attribute, use the IASHAR selector with the MQINQ call.

TriggerControl (10-digit signed integer)

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:

TCOFF

Trigger messages not required.

No trigger messages are to be written for this queue. The value of *TriggerType* is irrelevant in this case.

TCON

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 IATRGC selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerData (64-byte character string)

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 CATRGD selector with the MQINQ call. To change the value of this attribute, use the MQSET call. The length of this attribute is given by LNTRGD.

TriggerDepth (10-digit signed integer)

Trigger depth.

This is the number of messages of priority *TriggerMsgPriority* or greater that must be on the queue before a trigger message is written. This applies when *TriggerType* is set to TTDPTH. The value of *TriggerDepth* is one or greater. This attribute is not used otherwise.

To determine the value of this attribute, use the IATRGD selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerMsgPriority (10-digit signed integer)

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 “Chapter 43. Attributes for the queue manager” on page 323); a value of zero causes all messages to contribute to the generation of trigger messages.

To determine the value of this attribute, use the IATRGP selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

TriggerType (10-digit signed integer)

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:

TTNONE

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 TCOFF.

TTFRST

Trigger message when queue depth goes from 0 to 1.

A trigger message is written whenever the number of messages of priority *TriggerMsgPriority* or greater on the queue changes from 0 to 1.

TTEVRY

Trigger message for every message.

A trigger message is written whenever a message of priority *TriggerMsgPriority* or greater arrives on the queue.

TTDPTH

Trigger message when depth threshold exceeded.

A trigger message is written whenever the number of messages of priority *TriggerMsgPriority* or greater on the queue equals or exceeds *TriggerDepth*. After the trigger message has been written, *TriggerControl* is set to TCOFF to prevent further triggering until it is explicitly turned on again.

Attributes - local and model queues

To determine the value of this attribute, use the IATRGT selector with the MQINQ call. To change the value of this attribute, use the MQSET call.

Usage (10-digit signed integer)
Queue usage.

This indicates what the queue is used for. It is one of the following:

USNORM

Normal usage.

This is a queue that normal applications use when putting and getting messages; the queue is not a transmission queue.

USTRAN

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 OOOOUT 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 “Chapter 21. MQXQH - Transmission queue header” on page 193), otherwise errors may occur during the transmission process. Context is not passed or set unless one of the PM* context options is specified.

To determine the value of this attribute, use the IAUSAG selector with the MQINQ call.

Chapter 39. 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 51. Attributes for local definitions of remote queues

Attribute	Description	Page
<i>RemoteQMgrName</i>	Name of remote queue manager	313
<i>RemoteQName</i>	Name of remote queue	314
<i>XmitQName</i>	Transmission queue name	314

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 (48-byte character string)

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 *MDRM*.

Note: No validation is performed on the value specified for this attribute when the queue definition is created or modified.

Attributes - remote queues

To determine the value of this attribute, use the CARQMN selector with the MQINQ call.

The length of this attribute is given by LNQMNM.

RemoteQName (48-byte character string)

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 *MDRQ*.

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 CARQN selector with the MQINQ call.

The length of this attribute is given by LNQN.

XmitQName (48-byte character string)

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 CAXQN selector with the MQINQ call.

The length of this attribute is given by LNQN.

Chapter 40. Attributes for alias queues

The following attribute is associated with alias queues:

BaseQName (48-byte character string)

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:

QTLOC

Local queue.

QTREM

Local definition of a remote queue.

QTCLUS

Cluster queue.

To determine the value of this attribute, use the CABASQ selector with the MQINQ call.

The length of this attribute is given by LNQN.

Attributes - alias queues

Chapter 41. Attributes for namelists

Namelists are supported in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

The following table summarizes the attributes that are specific to namelists. The attributes are described in alphabetic order.

Table 52. Attributes for namelists

Attribute	Description	Page
<i>AlterationDate</i>	Date when definition was last changed	317
<i>AlterationTime</i>	Time when definition was last changed	317
<i>NameCount</i>	Number of names in namelist	317
<i>NamelistDesc</i>	Namelist description	317
<i>NamelistName</i>	Namelist name	318
<i>Names</i>	A list of <i>NameCount</i> names	318

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

NameCount (10-digit signed integer)

Number of names in namelist.

This is greater than or equal to zero. The following value is defined:

NCMXNL

Maximum number of names in a namelist.

To determine the value of this attribute, use the IANAMC selector with the MQINQ call.

NamelistDesc (64-byte character string)

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

Attributes - namelists

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 CALSTD selector with the MQINQ call.

The length of this attribute is given by LNNLD.

NameListName (48-byte character string)
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 CALSTN selector with the MQINQ call.

The length of this attribute is given by LNNLN.

Names (48-byte character string×*NameCount*)
A list of *NameCount* names.

Each name is the name of an object that is defined to the local queue manager. For more information about object names, see the *MQSeries Application Programming Guide*.

To determine the value of this attribute, use the CANAMS selector with the MQINQ call.

The length of each name in the list is given by LNOBJN.

Chapter 42. Attributes for process definitions

The following table summarizes the attributes that are specific to process definitions. The attributes are described in alphabetic order.

Table 53. Attributes for process definitions

Attribute	Description	Page
<i>AlterationDate</i>	Date when definition was last changed	319
<i>AlterationTime</i>	Time when definition was last changed	319
<i>AppId</i>	Application identifier	319
<i>AppType</i>	Application type	320
<i>EnvData</i>	Environment data	320
<i>ProcessDesc</i>	Process description	320
<i>ProcessName</i>	Process name	321
<i>UserData</i>	User data	321

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

AppId (256-byte character string)

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 meaning of *AppId* is determined by the trigger-monitor application. The trigger monitor provided by MQSeries requires *AppId* to be the name of an executable program.

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 CAAPPI selector with the MQINQ call. The length of this attribute is given by LNPROA.

Attributes - process definitions

AppType (10-digit signed integer)
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.

AppType 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 ATUFST through ATULST:

ATCICS

CICS transaction.

AT400 OS/400 application.

ATUFST

Lowest value for user-defined application type.

ATULST

Highest value for user-defined application type.

To determine the value of this attribute, use the IAAPPT selector with the MQINQ call.

EnvData (128-byte character string)
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.

The meaning of *EnvData* is determined by the trigger-monitor application. The trigger monitor provided by MQSeries appends *EnvData* to the parameter list passed to the started application. The parameter list consists of the MQTMC2 structure, followed by one blank, followed by *EnvData* with trailing blanks removed.

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 CAENVD selector with the MQINQ call. The length of this attribute is given by LNPROE.

ProcessDesc (64-byte character string)
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.

Attributes - process definitions

To determine the value of this attribute, use the CAPROD selector with the MQINQ call.

The length of this attribute is given by LNPROD.

ProcessName (48-byte character string)

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 CAPRON selector with the MQINQ call.

The length of this attribute is given by LNPRON.

UserData (128-byte character string)

User data.

This is a character string that contains user 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, 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 meaning of *UserData* is determined by the trigger-monitor application. The trigger monitor provided by MQSeries simply passes *UserData* to the started application as part of the parameter list. The parameter list consists of the MQTMC2 structure (containing *UserData*), followed by one blank, followed by *EnvData* with trailing blanks removed.

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 CAUSRD selector with the MQINQ call. The length of this attribute is given by LNPROU.

Attributes - process definitions

Chapter 43. 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 54. Attributes for the queue manager

Attribute	Description	Page
<i>AlterationDate</i>	Date when definition was last changed	324
<i>AlterationTime</i>	Time when definition was last changed	324
<i>AuthorityEvent</i>	Controls whether authorization (Not Authorized) events are generated	324
<i>ChannelAutoDef</i>	Controls whether automatic channel definition is permitted	324
<i>ChannelAutoDefEvent</i>	Controls whether channel automatic-definition events are generated	325
<i>ChannelAutoDefExit</i>	Name of user exit for automatic channel definition	325
<i>ClusterWorkloadData</i>	User data for cluster workload exit	325
<i>ClusterWorkloadExit</i>	Name of user exit for cluster workload management	325
<i>ClusterWorkloadLength</i>	Maximum length of message data passed to cluster workload exit	326
<i>CodedCharSetId</i>	Coded character set identifier	326
<i>CommandInputQName</i>	Command input queue name	326
<i>CommandLevel</i>	Command level	326
<i>DeadLetterQName</i>	Name of dead-letter queue	327
<i>DefXmitQName</i>	Default transmission queue name	328
<i>DistLists</i>	Distribution list support	328
<i>InhibitEvent</i>	Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated	329
<i>LocalEvent</i>	Controls whether local error events are generated	329
<i>MaxHandles</i>	Maximum number of handles	329
<i>MaxMsgLength</i>	Maximum message length in bytes	330
<i>MaxPriority</i>	Maximum priority	330
<i>MaxUncommittedMsgs</i>	Maximum number of uncommitted messages within a unit of work	330
<i>PerformanceEvent</i>	Controls whether performance-related events are generated	331
<i>Platform</i>	Platform on which the queue manager is running	331
<i>QMGrDesc</i>	Queue manager description	331
<i>QMGrIdentifier</i>	Unique internally-generated identifier of queue manager	332
<i>QMGrName</i>	Queue manager name	332
<i>RemoteEvent</i>	Controls whether remote error events are generated	332

Attributes - queue manager

Table 54. Attributes for the queue manager (continued)

Attribute	Description	Page
<i>RepositoryName</i>	Name of cluster for which this queue manager provides repository services	332
<i>RepositoryNameList</i>	Name of namelist object containing names of clusters for which this queue manager provides repository services	332
<i>StartStopEvent</i>	Controls whether start and stop events are generated	333
<i>SyncPoint</i>	Syncpoint availability	333
<i>TriggerInterval</i>	Trigger-message interval	333

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 OTQM object, and using the MQINQ call with the handle returned. They can also all be displayed with the DISPLAY QMGR command.

AlterationDate (12-byte character string)

Date when definition was last changed.

This is the date when the definition was last changed. The format of the date is YYYY-MM-DD, padded with two trailing blanks to make the length 12 bytes.

To determine the value of this attribute, use the CAALTD selector with the MQINQ call. The length of this attribute is given by LNDATE.

AlterationTime (8-byte character string)

Time when definition was last changed.

This is the time when the definition was last changed. The format of the time is HH.MM.SS.

To determine the value of this attribute, use the CAALTT selector with the MQINQ call. The length of this attribute is given by LNTIME.

AuthorityEvent (10-digit signed integer)

Controls whether authorization (Not Authorized) events are generated.

It is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IAAUTE selector with the MQINQ call.

ChannelAutoDef (10-digit signed integer)

Controls whether automatic channel definition is permitted.

This attribute controls the automatic definition of channels of type CTRCVR and CTSVCN. Note that the automatic definition of CTCLSD channels is always enabled. The value is one of the following:

CHADDI

Channel auto-definition disabled.

CHADEN

Channel auto-definition enabled.

To determine the value of this attribute, use the IACAD selector with the MQINQ call.

ChannelAutoDefEvent (10-digit signed integer)

Controls whether channel automatic-definition events are generated.

This applies to channels of type CTRCVR, CTSVCN, and CTCLSD. The value is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IACADE selector with the MQINQ call.

ChannelAutoDefExit (n-byte character string)

Name of user exit for automatic channel definition.

If this name is nonblank, and *ChannelAutoDef* has the value CHADEN, the exit is called each time that the queue manager is about to create a channel definition. This applies to channels of type CTRCVR, CTSVCN, and CTCLSD. 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 CACADX selector with the MQINQ call. The length of this attribute is given by LNEXN.

ClusterWorkloadData (32-byte character string)

User data for cluster workload exit.

This is a user-defined 32-byte character string that is passed to the cluster workload exit when it is called. If there is no data to pass to the exit, the string is blank.

To determine the value of this attribute, use the CACLWD selector with the MQINQ call.

ClusterWorkloadExit (n-byte character string)

Name of user exit for cluster workload management.

If this name is nonblank, the exit is called each time that a message is put to a cluster queue or moved from one cluster-sender queue to another. The exit can

Attributes - queue manager

then decide whether to accept the queue instance selected by the queue manager as the destination for the message, or choose another queue instance.

Note: Both the length and the value of this attribute are environment specific. See the *MQSeries Intercommunication* manual for details of the value of this attribute in various environments.

To determine the value of this attribute, use the CACLWX selector with the MQINQ call. The length of this attribute is given by LNEXN.

ClusterWorkloadLength (10-digit signed integer)

Maximum length of message data passed to cluster workload exit.

This is the maximum length of message data that is passed to the cluster workload exit. The actual length of data passed to the exit is the minimum of the following:

- The length of the message.
- The queue-manager's *MaxMsgLength* attribute.
- The *ClusterWorkloadLength* attribute.

To determine the value of this attribute, use the IACLWL selector with the MQINQ call.

CodedCharSetId (10-digit signed integer)

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 OS/400, the value is that which is set in the environment when the queue manager is first created.

To determine the value of this attribute, use the IACCSI selector with the MQINQ call.

CommandInputQName (48-byte character string)

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 OS/400, 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 CMESC. Refer to the *MQSeries Programmable System Management* book for details of the Escape command.

To determine the value of this attribute, use the CACMDQ selector with the MQINQ call. The length of this attribute is given by LNQN.

CommandLevel (10-digit signed integer)

Command Level.

This indicates the level of system control commands supported by the queue manager. The value is one of the following:

CMLVL1

Level 1 of system control commands.

This value is returned by the following:

- MQSeries for OS/400
 - Version 2 Release 3
 - Version 3 Release 1
 - Version 3 Release 6

CML320

Level 320 of system control commands.

This value is returned by the following:

- MQSeries for OS/400
 - Version 3 Release 2
 - Version 3 Release 7

CML420

Level 420 of system control commands.

This value is returned by the following:

- MQSeries for AS/400
 - Version 4 Release 2.0
 - Version 4 Release 2.1

CML510

Level 510 of system control commands.

This value is returned by the following:

- MQSeries for AS/400 Version 5 Release 1

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 IACMDL selector with the MQINQ call.

DeadLetterQName (48-byte character string)

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.

Attributes - queue manager

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 *MDRQ* 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 *MDEXP* field described in “Chapter 10. MQMD - Message descriptor” on page 83) are **not** transferred to this queue when they are discarded. However, an expiration report message (ROEXP) is still generated and sent to the *MDRQ* 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 “Chapter 7. MQDLH - Dead-letter header” on page 43 for more details of this structure.

This queue must be a local queue, with a *Usage* attribute of USNORM.

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 CADLQ selector with the MQINQ call. The length of this attribute is given by LNQN.

DefXmitQName (48-byte character string)

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 CADXQN selector with the MQINQ call. The length of this attribute is given by LNQN.

DistLists (10-digit signed integer)

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:

DLSUPP

Distribution lists supported.

DLNSUP

Distribution lists not supported.

To determine the value of this attribute, use the IADIST selector with the MQINQ call.

InhibitEvent (10-digit signed integer)

Controls whether inhibit (Inhibit Get and Inhibit Put) events are generated.

It is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IAINHE selector with the MQINQ call.

LocalEvent (10-digit signed integer)

Controls whether local error events are generated.

It is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IALCLE selector with the MQINQ call.

MaxHandles (10-digit signed integer)

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.

The value is in the range 1 through 999 999 999. The default value is determined by the environment:

- On OS/400, the default value is 256.

To determine the value of this attribute, use the IAMHND selector with the MQINQ call.

Attributes - queue manager

MaxMsgLength (10-digit signed integer)
Maximum message length in bytes.

This is the length of the longest *physical* message that can be handled by the queue manager. However, because the *MaxMsgLength* queue-manager attribute can be set independently of the *MaxMsgLength* local-queue attribute, 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 *logical* message that is longer than the lesser of the two *MaxMsgLength* attributes, but only if the application specifies the MFSEGA flag in MQMD. If that flag is specified, the upper limit for the length of a logical message is 999 999 999 bytes, but usually resource constraints imposed by the operating system, or by the environment in which the application is running, will result in a lower limit.

The lower limit for the *MaxMsgLength* attribute is 32 KB (32 768 bytes). The upper limit is determined by the environment:

- On OS/400, the maximum message length is 100 MB (104 857 600 bytes).

To determine the value of this attribute, use the IAMLEN selector with the MQINQ call.

MaxPriority (10-digit signed integer)
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 IAMPRI selector with the MQINQ call.

MaxUncommittedMsgs (10-digit signed integer)
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 PMSYP option
- Messages retrieved by the application with the GMSYP option
- Trigger messages and COA report messages generated by the queue manager for messages put with the PMSYP option
- COD report messages generated by the queue manager for messages retrieved with the GMSYP 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 GMSYP)
- Event messages generated by the queue manager (even if the call causing the event message specified PMSYP or GMSYP)

Notes:

1. 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.
2. When a message or segment is put with the PMSYP option, the number of uncommitted messages is incremented by one regardless of how many physical messages actually result from the put. (More than one physical message may result if the queue manager needs to subdivide the message or segment.)
3. When a distribution list is put with the PMSYP option, the number of uncommitted messages is incremented by one *for each physical message that is generated*. This can be as small as one, or as great as the number of destinations in the distribution list.

The lower limit for this attribute is 1; the upper limit is 999 999 999.

To determine the value of this attribute, use the IAMUNC selector with the MQINQ call.

PerformanceEvent (10-digit signed integer)

Controls whether performance-related events are generated.

It is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IAPFME selector with the MQINQ call.

Platform (10-digit signed integer)

Platform on which the queue manager is running.

This indicates the architecture of the platform on which the queue manager is running. The value is:

PL400 OS/400.

QMGrDesc (64-byte character string)

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 OS/400, the default value is blanks.

Attributes - queue manager

To determine the value of this attribute, use the CAQMD selector with the MQINQ call. The length of this attribute is given by LNQMMD.

QMgrIdentifier (48-byte character string)

Unique internally-generated identifier of queue manager.

This is an internally-generated unique name for the queue manager.

To determine the value of this attribute, use the CAQMID selector with the MQINQ call. The length of this attribute is given by LNQMID.

QMgrName (48-byte character string)

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 *MDMID* field described in “Chapter 10. MQMD - Message descriptor” on page 83). 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.

To determine the value of this attribute, use the CAQMN selector with the MQINQ call. The length of this attribute is given by LNQMNM.

RemoteEvent (10-digit signed integer)

Controls whether remote error events are generated.

It is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IARMTE selector with the MQINQ call.

RepositoryName (48-byte character string)

Name of cluster for which this queue manager provides repository services.

This is the name of a cluster for which this queue manager provides a repository-manager service. If the queue manager provides this service for more than one cluster, *RepositoryNameList* specifies the name of a namelist object that identifies the clusters, and *RepositoryName* is blank. At least one of *RepositoryName* and *RepositoryNameList* must be blank.

To determine the value of this attribute, use the CARPN selector with the MQINQ call. The length of this attribute is given by LNQMNM.

RepositoryNameList (48-byte character string)

Name of namelist object containing names of clusters for which this queue manager provides repository services.

This is the name of a namelist object that contains the names of clusters for which this queue manager provides a repository-manager service. If the queue manager provides this service for only one cluster, the namelist object contains

only one name. Alternatively, *RepositoryName* can be used to specify the name of the cluster, in which case *RepositoryNameList* is blank. At least one of *RepositoryName* and *RepositoryNameList* must be blank.

To determine the value of this attribute, use the CARPNL selector with the MQINQ call. The length of this attribute is given by LNNLN.

StartStopEvent (10-digit signed integer)

Controls whether start and stop events are generated.

It is one of the following:

EVRDIS

Event reporting disabled.

EVRENA

Event reporting enabled.

To determine the value of this attribute, use the IASSE selector with the MQINQ call.

SyncPoint (10-digit signed integer)

Syncpoint availability.

This indicates whether the local queue manager supports units of work and syncpointing with the MQGET, MQPUT, and MQPUT1 calls.

SPAVL

Units of work and syncpointing available.

SPNAVL

Units of work and syncpointing not available.

To determine the value of this attribute, use the IASYNC selector with the MQINQ call.

TriggerInterval (10-digit signed integer)

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 TTFRST. 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 TTFRST 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 IATRGI selector with the MQINQ call.

Attributes - queue manager

Part 4. Applications

Chapter 44. Building your application	337
MQSeries copy files	337
Preparing your programs to run	337
Interfaces to the AS/400 external syncpoint manager	338
Syncpoints in CICS for AS/400 applications	339
Chapter 45. Sample programs	341
Features demonstrated in the sample programs	342
Preparing and running the sample programs	343
Running the sample programs.	343
The Put sample program	343
Design of the Put sample program	344
The Browse sample program	344
Design of the Browse sample program	344
The Get sample program	345
Design of the Get sample program	345
The Request sample program	346
Using triggering with the Request sample.	346
Design of the Request sample program.	347
The Echo sample program	348
Design of the Echo sample program.	349
The Inquire sample program	349
Design of the Inquire sample program	350
The Set sample program.	351
Design of the Set sample program	351
The Triggering sample programs	352
The AMQ3TRG4 sample trigger monitor	352
Design of the trigger monitor	352
The AMQ3SRV4 sample trigger server	352
Design of the trigger server	353
Ending the Triggering sample programs	353
Running the samples using remote queues	353

Applications

Chapter 44. Building your application

The OS/400 publications describe how to build executable applications from the programs you write. This chapter describes the additional tasks, and the changes to the standard tasks, you must perform when building MQSeries for AS/400 applications to run under OS/400.

In addition to coding the MQI calls in your source code, you must add the appropriate language statements to include the MQSeries for AS/400 copy files for the RPG language. You should make yourself familiar with the contents of these files—their names, and a brief description of their contents are given in the following text.

MQSeries copy files

MQSeries for AS/400 provides copy files to assist you with writing your applications in the RPG programming language. They are suitable for use with the IBM ILE RPG/400™ Compiler (5716-RG1).

The copy files that MQSeries for AS/400 provides to assist with the writing of channel exits are described in the *MQSeries Intercommunication* book.

The names of the MQSeries for AS/400 copy files for RPG have the prefix CMQ. They have a suffix of G. There are separate copy files containing the named constants, and one file for each of the structures. The copy files are listed in Table 2 on page 8.

Note: For ILE RPG/400 they are supplied as members of file QRPGLESRC in library QMQM.

The structure declarations do not contain **DS** statements. This allows the application to declare a data structure (or a multiple-occurrence data structure) by coding the **DS** statement and using the **/COPY** statement to copy in the remainder of the declaration:

For ILE RPG/400 the statement is:

```
D*..1.....2.....3.....4.....5.....6.....7
D* Declare an MQMD data structure
D MQMD      DS
D/COPY CMQMDG
```

Preparing your programs to run

To create an executable MQSeries for AS/400 application, you have to compile the source code you have written.

To do this for ILE RPG/400, you can use the usual OS/400 commands, CRTRPGMOD and CRTPGM.

After creating your *MODULE, you also need to reference the service program QMQM/AMQZSTUB when creating your ILE RPG/400 program using the CRTPGM command.

Preparing programs

Make sure that the library containing the copy files (QMQM) is in the library list when you perform the compilation. QMQM *must* also be in the library list when you run the application.

Interfaces to the AS/400 external syncpoint manager

MQSeries for AS/400 uses native OS/400 commitment control as an external syncpoint coordinator. See the *AS/400 Programming: Backup and Recovery Guide* for more information about the commitment control capabilities of OS/400.

To start the OS/400 commitment control facilities, use the STRCMTCTL system command. To end commitment control, use the ENDCMTCTL system command.

Note: The default value of *Commitment definition scope* is *ACTGRP. This must be defined as *JOB for MQSeries for AS/400. For example:

```
STRCMTCTL LCKLVL(*ALL) CMTSCOPE(*JOB)
```

If you call MQPUT, MQPUT1, or MQGET, specifying MQPMO_SYNCPOINT or MQGMO_SYNCPOINT, after starting commitment control, MQSeries for AS/400 adds itself as an API commitment resource to the commitment definition. This is typically the first such call in a job. While there are any API commitment resources registered under a particular commitment definition, you cannot end commitment control for that definition.

MQSeries for AS/400 removes its registration as an API commitment resource when you disconnect from the queue manager, provided there are no pending MQI operations in the current unit of work.

If you disconnect from the queue manager while there are pending MQPUT, MQPUT1, or MQGET operations in the current unit of work, MQSeries for AS/400 remains registered as an API commitment resource so that it is notified of the next commit or rollback. When the next syncpoint is reached, MQSeries for AS/400 commits or rolls back the changes as required. It is possible for an application to disconnect and reconnect to a queue manager during an active unit of work and perform further MQGET and MQPUT operations inside the same unit of work (this is a pending disconnect).

If you attempt to issue an ENDCMTCTL system command for that commitment definition, message CPF8355 is issued, indicating that pending changes were active. This message also appears in the job log when the job ends. To avoid this, ensure that you commit or roll back all pending MQSeries for AS/400 operations, and that you disconnect from the queue manager. Thus, using COMMIT or ROLLBACK commands before ENDCMTCTL should enable end-commitment control to complete successfully.

When OS/400 commitment control is used as an external syncpoint coordinator, MQCMIT, MQBACK, and MQBEGIN calls may not be issued. Calls to these functions fail with the reason code MQRC_ENVIRONMENT_ERROR.

To commit or roll back (that is, to back out) your unit of work, use one of the programming languages that supports the commitment control. For example:

- CL commands: COMMIT and ROLLBACK
- ILE C Programming Functions: _Rcommit and _Rrollback
- RPG/400: COMIT and ROLBK
- COBOL/400: COMMIT and ROLLBACK

Syncpoints in CICS for AS/400 applications

MQSeries for AS/400 participates in units of work with CICS. You can use the MQI within a CICS application to put and get messages inside the current unit of work.

You can use the EXEC CICS SYNCPOINT command to establish a syncpoint that includes the MQSeries for AS/400 operations. To back out all changes up to the previous syncpoint, you can use the EXEC CICS SYNCPOINT ROLLBACK command.

If you use MQPUT, MQPUT1, or MQGET with the MQPMO_SYNCPOINT, or MQGMO_SYNCPOINT, option set in a CICS application, you cannot log off CICS until MQSeries for AS/400 has removed its registration as an API commitment resource. Therefore, you should commit or back out any pending put or get operations before you disconnect from the queue manager. This will allow you to log off CICS.

Applications

Chapter 45. Sample programs

This chapter describes the sample programs delivered with MQSeries for AS/400 for RPG. The samples demonstrate typical uses of the Message Queue Interface (MQI).

The samples are not intended to demonstrate general programming techniques, so some error checking that you may want to include in a production program has been omitted. However, these samples are suitable for use as a base for your own message queuing programs.

The source code for all the samples is provided with the product; this source includes comments that explain the message queuing techniques demonstrated in the programs.

There are two sets of ILE sample programs:

1. Programs using the MQI through a call to QMQM

The source exists in QMQMSAMP/QRPGLESRC. The members are named AMQ2xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name has a suffix of "R".

2. Programs using prototyped calls to the MQI

The source exists in QMQMSAMP/QRPGLESRC. The members are named AMQ3xxx4, where xxx indicates the sample function. Copy members exist in QMQM/QRPGLESRC. Each member name has a suffix of "G".

Notes:

- a. In Table 55 and Table 56 on page 342 the character "n" is used to represent the appropriate numeric value.
- b. Sample trigger programs only exist for programs using prototyped calls.

Table 55 gives a complete list of the sample programs delivered with MQSeries for AS/400 V3R1 or later, and shows the names of the programs in each of the supported programming languages. Notice that their names all start with the prefix AMQ, the fourth character in the name indicates the programming language.

Note: This chapter tells you how to use the ILE RPG/400 compiler. with prototyped calls to the MQI.

Table 55. Names of the sample programs

	RPG (ILE)
Put samples	AMQnPUT4
Browse samples	AMQnGBR4
Get samples	AMQnGET4
Request samples	AMQnREQ4
Echo samples	AMQnECH4
Inquire samples	AMQnINQ4
Set samples	AMQnSET4
Trigger Monitor sample	AMQ3TRG4

Sample programs

Table 55. Names of the sample programs (continued)

	RPG (ILE)
Trigger Server sample	AMQ3SRV4

In addition to these, the MQSeries for AS/400 sample option includes a sample data file, AMQSDATA, which can be used as input to certain sample programs. and sample CL programs that demonstrate administration tasks. The CL samples are described in the *MQSeries for AS/400 V5.1 System Administration*. You could use the sample CL program to create queues to use with the sample programs described in this chapter.

For information on how to run the sample programs, see “Preparing and running the sample programs” on page 343.

Features demonstrated in the sample programs

Table 56 shows the techniques demonstrated by the MQSeries for AS/400 sample programs. Some techniques occur in more than one sample program, but only one program is listed in the table. All the samples open and close queues using the MQOPEN and MQCLOSE calls, so these techniques are not listed separately in the table.

Table 56. Sample programs demonstrating use of the MQI

Technique	RPG (ILE)
Using the MQCONN and MQDISC calls	AMQnECH4 or AMQnINQ4
Implicitly connecting and disconnecting	AMQnPUT4
Putting messages using the MQPUT call	AMQnPUT4
Putting a single message using the MQPUT1 call	AMQnECH4 or AMQnINQ4
Replying to a request message	AMQnINQ4
Getting messages (no wait)	AMQnGBR4
Getting messages (wait with a time limit)	AMQnGET4
Getting messages (with data conversion)	AMQnECH4
Browsing a queue	AMQnGBR4
Using a shared input queue	AMQnINQ4
Using an exclusive input queue	AMQnREQ4
Using the MQINQ call	AMQnINQ4
Using the MQSET call	AMQnSET4
Using a reply-to queue	AMQnREQ4
Requesting exception messages	AMQnREQ4
Accepting a truncated message	AMQnGBR4
Using a resolved queue name	AMQnGBR4
Trigger processing	AMQ3SRV4 or AMQ3TRG4

Note: All the sample programs produce a spool file that contains the results of the processing.

Preparing and running the sample programs

Before you can run the MQSeries for AS/400 sample programs, you must compile them as you would any other MQSeries for AS/400 applications. To do this, you can use the OS/400 commands CRTRPGMOD and CRTPGM.

When you create the AMQ3xxx4 programs, you need to specify BNDSRVPGM(QMQM/LIBMQM) in the CRTPGM command. This includes the various MQ procedures in your program.

The sample programs are provided in library QMQMSAMP as members of QRPGLESRC. They use the copy files provided in library QMQM, so make sure this library is in the library list when you compile them. The RPG compiler gives information messages because the samples do not use many of the variables that are declared in the copy files.

Running the sample programs

You can use your own queues when you run the samples, or you can run AMQSAMP4 to create some sample queues. The source for this program is shipped in file QCLSRC in library QMQMSAMP. It can be compiled using the CRTCLPGM command.

To call one of the sample programs, use a command like:

```
CALL PGM(QMQMSAMP/AMQ3PUT4) PARM('Queue Name')
```

where Queue Name *must* be 48 characters in length, which you achieve by padding the queue name with the required number of blanks.

Note that for the Inquire and Set sample programs, the sample definitions created by AMQSAMP4 cause the C versions of these samples to be triggered. If you want to trigger the RPG versions, you must change the process definitions SYSTEM.SAMPLE.ECHOPROCESS and SYSTEM.SAMPLE.INQPROCESS and SYSTEM.SAMPLE.SETPROCESS. You can use the CHGMQMPCRC command (described in the *MQSeries for AS/400 V5.1 System Administration* book) to do this, or edit and run AMQSAMP4 with the alternative definition.

The Put sample program

The Put sample program, AMQnPUT4, puts messages on a queue using the MQPUT call.

To start the program, call the program and give the name of your target queue as a program parameter. The program puts a set of fixed messages on the queue; these messages are taken from the data block at the end of the program source code. A sample put program is AMQ3PUT4 in library QMQMSAMP.

Using this example program, the command is:

```
CALL PGM(QMQMSAMP/AMQ3PUT4) PARM('Queue Name')
```

where Queue Name *must* be 48 characters in length, which you achieve by padding the queue name with the required number of blanks.

Put sample

Design of the Put sample program

The program uses the MQOPEN call with the OOOUT option to open the target queue for putting messages. The results are output to a spool file. If it cannot open the queue, the program writes an error message containing the reason code returned by the MQOPEN call. To keep the program simple, on this and on subsequent MQI calls, the program uses default values for many of the options.

For each line of data contained in the source code, the program reads the text into a buffer and uses the MQPUT call to create a datagram message containing the text of that line. The program continues until either it reaches the end of the input or the MQPUT call fails. If the program reaches the end of the input, it closes the queue using the MQCLOSE call.

The Browse sample program

The Browse sample program, AMQnGBR4, browses messages on a queue using the MQGET call.

The program retrieves copies of all the messages on the queue you specify when you call the program; the messages remain on the queue. You could use the supplied queue SYSTEM.SAMPLE.LOCAL; run the Put sample program first to put some messages on the queue. You could use the queue SYSTEM.SAMPLE.ALIAS, which is an alias name for the same local queue. The program continues until it reaches the end of the queue or an MQI call fails.

An example of a command to call the RPG program is:

```
CALL PGM(QMQMSAMP/AMQ3GBR4) PARM('Queue Name')
```

where Queue Name *must* be 48 characters in length, which you achieve by padding the queue name with the required number of blanks. Therefore, if you are using SYSTEM.SAMPLE.LOCAL as your target queue, you will need 29 blank characters.

Design of the Browse sample program

The program opens the target queue using the MQOPEN call with the OOBROW option. If it cannot open the queue, the program writes an error message to its spool file, containing the reason code returned by the MQOPEN call.

For each message on the queue, the program uses the MQGET call to copy the message from the queue, then displays the data contained in the message. The MQGET call uses these options:

GMBRWN

After the MQOPEN call, the browse cursor is positioned logically before the first message in the queue, so this option causes the *first* message to be returned when the call is first made.

GMNWT

The program does not wait if there are no messages on the queue.

GMATM

The MQGET call specifies a buffer of fixed size. If a message is longer than this buffer, the program displays the truncated message, together with a warning that the message has been truncated.

The program demonstrates how you must clear the *MDMID* and *MDCID* fields of the MQMD structure after each MQGET call because the call sets these fields to the

values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The program continues to the end of the queue; at this point the MQGET call returns the RC2033 (no message available) reason code and the program displays a warning message. If the MQGET call fails, the program writes an error message that contains the reason code in its spool file.

The program then closes the queue using the MQCLOSE call.

The Get sample program

The Get sample program, AMQnGET4, gets messages from a queue using the MQGET call.

When the program is called, it removes messages from the specified queue. You could use the supplied queue SYSTEM.SAMPLE.LOCAL; run the Put sample program first to put some messages on the queue. You could use the SYSTEM.SAMPLE.ALIAS queue, which is an alias name for the same local queue. The program continues until the queue is empty or an MQI call fails.

An example of a command to call the RPG program is:

```
CALL PGM(QMQMSAMP/AMQ3GET4) PARM('Queue Name')
```

where Queue Name *must* be 48 characters in length, which you achieve by padding the queue name with the required number of blanks. Therefore, if you are using SYSTEM.SAMPLE.LOCAL as your target queue, you will need 29 blank characters.

Design of the Get sample program

The program opens the target queue for getting messages; it uses the MQOPEN call with the OOINPQ option. If it cannot open the queue, the program writes an error message containing the reason code returned by the MQOPEN call in its spool file.

For each message on the queue, the program uses the MQGET call to remove the message from the queue; it then displays the data contained in the message. The MQGET call uses the GMWT option, specifying a wait interval (*GMWI*) of 15 seconds, so that the program waits for this period if there is no message on the queue. If no message arrives before this interval expires, the call fails and returns the RC2033 (no message available) reason code.

The program demonstrates how you must clear the *MDMID* and *MDCID* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

The MQGET call specifies a buffer of fixed size. If a message is longer than this buffer, the call fails and the program stops.

The program continues until either the MQGET call returns the RC2033 (no message available) reason code or the MQGET call fails. If the call fails, the program displays an error message that contains the reason code.

Get sample

The program then closes the queue using the MQCLOSE call.

The Request sample program

The Request sample program, AMQnREQ4, demonstrates client/server processing. The sample is the client that puts request messages on a queue that is processed by a server program. It waits for the server program to put a reply message on a reply-to queue.

The Request sample puts a series of request messages on a queue using the MQPUT call. These messages specify SYSTEM.SAMPLE.REPLY as the reply-to queue. The program waits for reply messages, then displays them. Replies are sent only if the target queue (which we will call the *server queue*) is being processed by a server application, or if an application is triggered for that purpose (the Inquire and Set sample programs are designed to be triggered). The sample waits 5 minutes for the first reply to arrive (to allow time for a server application to be triggered) and 15 seconds for subsequent replies, but it can end without getting any replies.

To start the program, call the program and give the name of your target queue as a program parameter. The program puts a set of fixed messages on the queue; these messages are taken from the data block at the end of the program source code.

Using triggering with the Request sample

To run the sample using triggering, start the trigger server program, AMQ3SRV4, against the required initiation queue in one job, then start AMQnREQ4 in another job. This means that the trigger server is ready when the Request sample program sends a message.

Notes:

1. The samples use the SYSTEM SAMPLE TRIGGER queue as the initiation queue for SYSTEM.SAMPLE.ECHO, SYSTEM.SAMPLE.INQ, or SYSTEM.SAMPLE.SET local queues. Alternatively, you can define your own initiation queue.
2. The sample definitions created by AMQSAMP4 cause the C version of the sample to be triggered. If you want to trigger the RPG version, you must change the process definitions SYSTEM.SAMPLE.ECHOPROCESS and SYSTEM.SAMPLE.INQPROCESS and SYSTEM.SAMPLE.SETPROCESS. You can use the CHGMQMPCRC command (described in the *MQSeries for AS/400 V5.1 System Administration*) to do this, or edit and run your own version of AMQSAMP4.
3. You need to compile the trigger server program from the source provided in QMQMSAMP/QRPGLESRC.

Depending on the trigger process you want to run, AMQ3REQ4 should be called with the parameter specifying request messages to be placed on one of these sample server queues:

- SYSTEM.SAMPLE.ECHO (for the Echo sample programs)
- SYSTEM.SAMPLE.INQ (for the Inquire sample programs)
- SYSTEM.SAMPLE.SET (for the Set sample programs)

A flow chart for the SYSTEM.SAMPLE.ECHO program is shown in Figure 1 on page 348. Using the example the command to issue the RPG program request to this server is:

```
CALL PGM(QMQMSAMP/AMQ3REQ4) PARM('SYSTEM.SAMPLE.ECHO  
+ 30 blank characters')
```


because the queue name *must* be 48 characters in length.

Note: This sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, server applications are not triggered by the messages you send.

If you want to attempt further examples, you can try the following variations:

- Use AMQ3TRG4 instead of AMQ3SRV4 to submit the job instead, but potential job submission delays could make it less easy to follow what is happening.
- Use the SYSTEM.SAMPLE.INQ and SYSTEM.SAMPLE.SET sample queues. Using the example data file the commands to issue the RPG program requests to these servers are, respectively:

```
CALL PGM(QMQMSAMP/AMQ3INQ4) PARM('SYSTEM.SAMPLE.INQ
+ 31 blank characters')
CALL PGM(QMQMSAMP/AMQ3SET4) PARM('SYSTEM.SAMPLE.SET
+ 31 blank characters')
```

because the queue name *must* be 48 characters in length.

These sample queues also have a trigger type of FIRST.

Design of the Request sample program

The program opens the server queue so that it can put messages. It uses the MQOPEN call with the OOOOUT option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

The program then opens the reply-to queue called SYSTEM.SAMPLE.REPLY so that it can get reply messages. For this, the program uses the MQOPEN call with the OOINPX option. If it cannot open the queue, the program displays an error message containing the reason code returned by the MQOPEN call.

For each line of input, the program then reads the text into a buffer and uses the MQPUT call to create a request message containing the text of that line. On this call the program uses the ROEXCD report option to request that any report messages sent about the request message will include the first 100 bytes of the message data. The program continues until either it reaches the end of the input or the MQPUT call fails.

The program then uses the MQGET call to remove reply messages from the queue, and displays the data contained in the replies. The MQGET call uses the GMWT option, specifying a wait interval (*GMWT*) of 5 minutes for the first reply (to allow time for a server application to be triggered) and 15 seconds for subsequent replies. The program waits for these periods if there is no message on the queue. If no message arrives before this interval expires, the call fails and returns the RC2033 (no message available) reason code. The call also uses the GMATM option, so messages longer than the declared buffer size are truncated.

The program demonstrates how you must clear the *MDMID* and *MDCOD* fields of the MQMD structure after each MQGET call because the call sets these fields to the values contained in the message it retrieves. Clearing these fields means that successive MQGET calls retrieve messages in the order in which the messages are held in the queue.

Get sample

The program continues until either the MQGET call returns the RC2033 (no message available) reason code or the MQGET call fails. If the call fails, the program displays an error message that contains the reason code.

The program then closes both the server queue and the reply-to queue using the MQCLOSE call. 57 shows the changes to the Echo sample program that are necessary to run the Inquire and Set sample programs.

Note: The details for the Echo sample program are included as a reference.

Table 57. Client/Server sample program details

Program name	SYSTEM/SAMPLE queue	Program started
Echo	ECHO	AMQ3ECH4
Inquire	INQ	AMQ3INQ4
Set	SET	AMQ3SET4

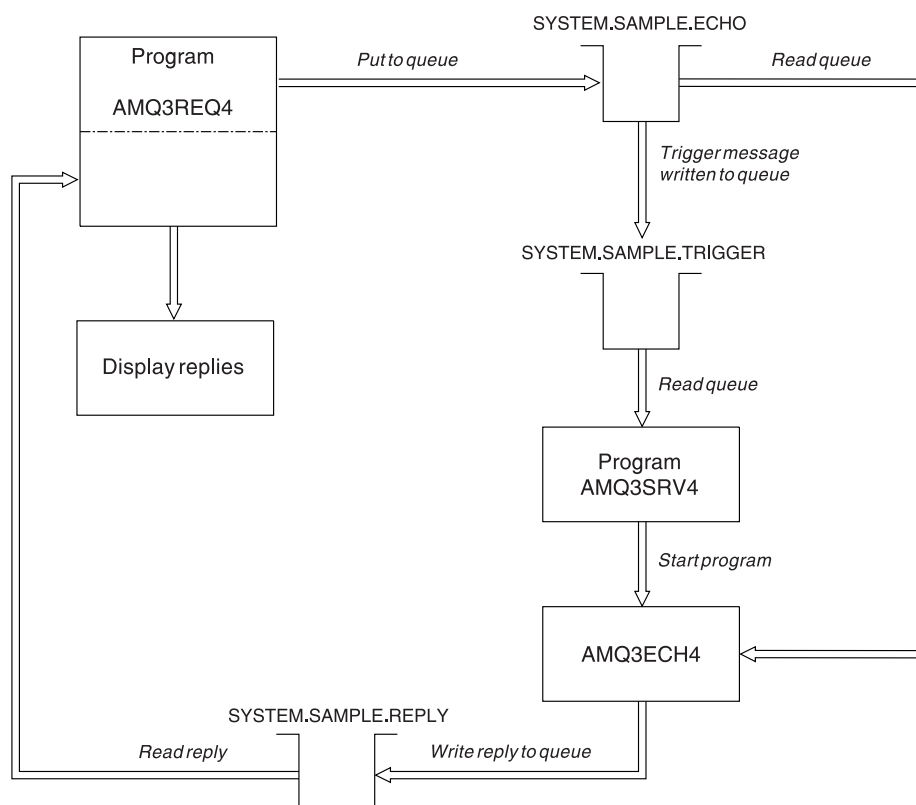


Figure 1. Sample Client/Server (Echo) program flowchart

The Echo sample program

The Echo sample programs return the message send to a reply queue. The program is named AMQnECH4

The programs are intended to run as triggered programs, so their only input is the data read from the queue named in the trigger message structure.

For the triggering process to work, you must ensure that the Echo sample program you want to use is triggered by messages arriving on queue

SYSTEM.SAMPLE.ECHO. To do this, specify the name of the Echo sample program you want to use in the *AppId* field of the process definition SYSTEM.SAMPLE.ECHOPROCESS. (For this, you can use the CHGMQMPCRC command, described in the *MQSeries for AS/400 V5.1 System Administration*.) The sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, the Echo sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQ3SRV4 in one job, then start AMQnREQ4 in another. You could use AMQ3TRG4 instead of AMQ3SRV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample programs to send messages to queue SYSTEM.SAMPLE.ECHO. The Echo sample programs send a reply message containing the data in the request message to the reply-to queue specified in the request message.

Design of the Echo sample program

When the program is triggered, it explicitly connects to the default queue manager using the MQCONN call. Although this is not necessary for MQSeries for AS/400, this means you could use the same program on other platforms without changing the source code.

The program then opens the queue named in the trigger message structure it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the GMATM and GMWT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program uses the MQPUT call to put a reply message on the reply-to queue. This message contains the contents of the request message.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

This program can also respond to messages sent to the queue from platforms other than MQSeries for AS/400, although no sample is supplied for this situation. To make the ECHO program work, you:

- Write a program, correctly specifying the *Format*, *Encoding*, and *CCSID* fields, to send text request messages.

The ECHO program requests the queue manager to perform message data conversion, if this is needed.

- Specify CONVERT(*YES) on the MQSeries for AS/400 sending channel, if the program you have written does not provide similar conversion for the reply.

The Inquire sample program

The Inquire sample program, AMQnINQ4, inquires about some of the attributes of a queue using the MQINQ call.

Inquire sample

The program is intended to run as a triggered program, so its only input is an MQTMC (trigger message) structure that contains the name of a target queue whose attributes are to be inquired.

For the triggering process to work, you must ensure that the Inquire sample program is triggered by messages arriving on queue SYSTEM.SAMPLE.INQ. To do this, specify the name of the Inquire sample program in the *ApplId* field of the SYSTEM.SAMPLE.INQPROCESS process definition. (For this, you can use the CHGMQMPRC command, described in the *MQSeries for AS/400 V5.1 System Administration* book.) The sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, the Inquire sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQ3SRV4 in one job, then start AMQnREQ4 in another. You could use AMQ3TRG4 instead of AMQ3SRV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample program to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.INQ. For each request message, the Inquire sample program sends a reply message containing information about the queue specified in the request message. The replies are sent to the reply-to queue specified in the request message.

Design of the Inquire sample program

When the program is triggered, it explicitly connects to the default queue manager using the MQCONN call. Although this is not necessary for MQSeries for AS/400, this means you could use the same program on other platforms without changing the source code.

The program then opens the queue named in the trigger message structure it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the GMATM and GMWT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program reads the name of the queue (which we will call the *target queue*) contained in the data and opens that queue using the MQOPEN call with the OOINQ option. The program then uses the MQINQ call to inquire about the values of the *InhibitGet*, *CurrentQDepth*, and *OpenInputCount* attributes of the target queue.

If the MQINQ call is successful, the program uses the MQPUT call to put a reply message on the reply-to queue. This message contains the values of the 3 attributes.

If the MQOPEN or MQINQ call is unsuccessful, the program uses the MQPUT call to put a *report* message on the reply-to queue. In the *MDFB* field of the message descriptor of this report message is the reason code returned by either the MQOPEN or MQINQ call, depending on which one failed.

After the MQINQ call, the program closes the target queue using the MQCLOSE call.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

The Set sample program

The Set sample program, AMQnSET4, inhibits put operations on a queue by using the MQSET call to change the queue's *InhibitPut* attribute.

The program is intended to run as a triggered program, so its only input is an MQTMC (trigger message) structure that contains the name of a target queue whose attributes are to be inquired.

For the triggering process to work, you must ensure that the Set sample program is triggered by messages arriving on queue SYSTEM.SAMPLE.SET. To do this, specify the name of the Set sample program in the *ApplId* field of the process definition SYSTEM.SAMPLE.SETPROCESS. (For this, you can use the CHGMQMPRC command, described in the *MQSeries for AS/400 V5.1 System Administration*.) The sample queue has a trigger type of FIRST, so if there are already messages on the queue before you run the Request sample, the Set sample is not triggered by the messages you send.

When you have set the definition correctly, first start AMQ3SRV4 in one job, then start AMQnREQ4 in another. You could use AMQ3TRG4 instead of AMQ3SRV4, but potential job submission delays could make it less easy to follow what is happening.

Use the Request sample program to send request messages, each containing just a queue name, to queue SYSTEM.SAMPLE.SET. For each request message, the Set sample program sends a reply message containing a confirmation that put operations have been inhibited on the specified queue. The replies are sent to the reply-to queue specified in the request message.

Design of the Set sample program

When the program is triggered, it explicitly connects to the default queue manager using the MQCONN call. Although this is not necessary for MQSeries for AS/400, this means you could use the same program on other platforms without changing the source code.

The program then opens the queue named in the trigger message structure it was passed when it started. (For clarity, we will call this the *request queue*.) The program uses the MQOPEN call to open this queue for shared input.

The program uses the MQGET call to remove messages from this queue. This call uses the GMATM and GMWT options, with a wait interval of 5 seconds. The program tests the descriptor of each message to see if it is a request message; if it is not, the program discards the message and displays a warning message.

For each request message removed from the request queue, the program reads the name of the queue (which we will call the *target queue*) contained in the data and opens that queue using the MQOPEN call with the OOSSET option. The program then uses the MQSET call to set the value of the *InhibitPut* attribute of the target queue to QAPUTI.

Set sample

If the MQSET call is successful, the program uses the MQPUT call to put a reply message on the reply-to queue. This message contains the string `PUT inhibited`.

If the MQOPEN or MQSET call is unsuccessful, the program uses the MQPUT call to put a *report* message on the reply-to queue. In the *MDFB* field of the message descriptor of this report message is the reason code returned by either the MQOPEN or MQSET call, depending on which one failed.

After the MQSET call, the program closes the target queue using the MQCLOSE call.

When there are no messages remaining on the request queue, the program closes that queue and disconnects from the queue manager.

The Triggering sample programs

MQSeries for AS/400 supplies two Triggering sample programs that are written in ILE/RPG. The programs are:

AMQ3TRG4

This is a trigger monitor for the OS/400 environment. It submits an OS/400 job for the application to be started, but this means there is a processing overhead associated with each trigger message.

AMQ3SRV4

This is a trigger server for the OS/400 environment. For each trigger message, this server runs the start command in its own job to start the specified application. The trigger server can call CICS transactions.

C language versions of these samples are also available as executable programs in library QMQM, called AMQSTRG4 and AMQSERV4.

The AMQ3TRG4 sample trigger monitor

AMQ3TRG4 is a trigger monitor. It takes one parameter: the name of the initiation queue it is to serve. AMQSAMP4 defines a sample initiation queue, `SYSTEM.SAMPLE.TRIGGER`, that you can use when you try the sample programs.

AMQ3TRG4 submits an OS/400 job for each valid trigger message it gets from the initiation queue.

Design of the trigger monitor

The trigger monitor opens the initiation queue and gets messages from the queue, specifying an unlimited wait interval.

The trigger monitor submits an OS/400 job to start the application specified in the trigger message, and passes an MQTMC (a character version of the trigger message) structure. The environment data in the trigger message is used as job submission parameters.

Finally, the program closes the initiation queue.

The AMQ3SRV4 sample trigger server

AMQ3SRV4 is a trigger server. It takes one parameter: the name of the initiation queue it is to serve. AMQSAMP4 defines a sample initiation queue, `SYSTEM.SAMPLE.TRIGGER`, that you can use when you try the sample programs.

For each trigger message, AMQ3SRV4 runs a start command in its own job to start the specified application.

Using the example trigger queue the command to issue is:

```
CALL PGM(QMQM/AMQ3SRV4) PARM('Queue Name')
```

where Queue Name *must* be 48 characters in length, which you achieve by padding the queue name with the required number of blanks. Therefore, if you are using SYSTEM.SAMPLE.TRIGGER as your target queue, you will need 28 blank characters.

Design of the trigger server

The design of the trigger server is similar to that of the trigger monitor, except the trigger server:

- Allows CICS as well as OS/400 applications
- Does not use the environment data from the trigger message
- Calls OS/400 applications in its own job (or uses STRCICSUSR to start CICS applications) rather than submitting an OS/400 job
- Opens the initiation queue for shared input, so many trigger servers can run at the same time

Note: Programs started by AMQ3SRV4 must not use the MQDISC call because this will stop the trigger server. If programs started by AMQ3SRV4 use the MQCONN call, they will get the RC2002 reason code.

Ending the Triggering sample programs

A trigger monitor program can be ended by the sysrequest option 2 (ENDRQS) or by inhibiting gets from the trigger queue. If the sample trigger queue is used the command is:

```
CHGMQM QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*NO)
```

Note: To start triggering again on this queue, you *must* enter the command:

```
CHGMQM QNAME('SYSTEM.SAMPLE.TRIGGER') GETENBL(*YES)
```

Running the samples using remote queues

You can demonstrate remote queuing by running the samples on connected message queue managers.

Program AMQSAMP4 provides a local definition of a remote queue (SYSTEM.SAMPLE.REMOTE) that uses a remote queue manager named OTHER. To use this sample definition, change OTHER to the name of the second message queue manager you want to use. You must also set up a message channel between your two message queue managers; for information on how to do this, see the *MQSeries Intercommunication* book.

The Request sample program puts its own local queue manager name in the *MDRM* field of messages it sends. The Inquire and Set samples send reply messages to the queue and message queue manager named in the *MDRQ* and *MDRM* fields of the request messages they process.

Applications

Part 5. Appendixes

Appendix A. Return codes

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 codes

The completion code parameter (*CMPCOD*) 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:

CCOK

Successful completion.

The call completed fully; all output parameters have been set. The *REASON* parameter always has the value RCNONE in this case.

CCWARN

Warning (partial completion).

The call completed partially. Some output parameters may have been set in addition to the *CMPCOD* and *REASON* output parameters. The *REASON* parameter gives additional information about the partial completion.

CCFAIL

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 *CMPCOD* 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 codes

The reason code parameter (*REASON*) is a qualification to the completion code parameter (*CMPCOD*).

If there is no special reason to report, RCNONE is returned. A successful call returns CCOK and RCNONE.

If the completion code is either CCWARN or CCFAIL, the queue manager always reports a qualifying reason; details are given under each call description.

Return codes

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 *DLREA* field of the MQDLH structure (for messages on the dead-letter queue)
- The *MDFB* 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.

RCNONE

(0, X'000') No reason to report.

The call completed normally. The completion code (*CMPCOD*) is CCOK.

Corrective action: None.

RC0900

(900, X'384') Lowest value for an application-defined reason code returned by a data-conversion exit.

Data-conversion exits can return reason codes in the range RC0900 through RC0999 to indicate particular conditions that the exit has detected.

Corrective action: As defined by the writer of the data-conversion exit.

RC0999

(999, X'3E7') Highest value for application-defined reason code returned by a data-conversion exit.

Data-conversion exits can return reason codes in the range RC0900 through RC0999 to indicate particular conditions that the exit has detected.

Corrective action: As defined by the writer of the data-conversion exit.

RC2001

(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, a local definition of a remote queue, or a cluster queue.

Corrective action: Correct the queue definitions.

RC2002

(2002, X'7D2') Application already connected.

An MQCONN call was issued, but the application is already connected to the queue manager.

Corrective action: None. The *HCONN* parameter returned has the same value as was returned for the previous MQCONN call.

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.

RC2003

(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 CCFAIL; the MQDISC call returns completion code CCWARN.
 - On OS/390, 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 OS/390 this case does not occur.

Corrective action: Check the returns from previous calls to the queue manager. For example, a previous MQPUT call may have failed.

RC2004

(2004, X'7D4') Buffer parameter not valid.

The *BUFFER* parameter is not valid for one of the following reasons:

- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)
- The parameter pointer points to storage that cannot be accessed for the entire length specified by *BUFLN*.
- For calls where *BUFFER* is an output parameter: the parameter pointer points to read-only storage.

Corrective action: Correct the parameter.

RC2005

(2005, X'7D5') Buffer length parameter not valid.

The *BUFLN* parameter is not valid, or the parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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 value that is zero or greater. For the mqAddString and mqSetString calls, the special value MQBL_NULL_TERMINATED is also valid.

RC2006

(2006, X'7D6') Length of character attributes not valid.

CALEN 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

Return codes

the parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Specify a value large enough to hold the concatenated strings for all selected attributes.

RC2007

(2007, X'7D7') Character attributes string not valid.

CHRATR 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 *CALEN*. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2008

(2008, X'7D8') Not enough space allowed for character attributes.

For MQINQ calls, *CALEN* is not large enough to contain all of the character attributes for which *CA** selectors are specified in the *SELS* parameter.

The call still completes, with the *CHRATR* 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.

Corrective action: Specify a large enough value, unless only a subset of the values is needed.

RC2009

(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 GMWT 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 *CMPCOD* of CCFAIL.

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.

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.

RC2010

(2010, X'7DA') Data length parameter not valid.

The *DATLEN* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

This reason can also be returned to an MQ client program that is putting and getting messages, 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.

RC2011

(2011, X'7DB') Name of dynamic queue not valid.

On the MQOPEN call, a model queue is specified in the *ODDN* field of the *OBJDSC* parameter, but the *ODDN* field is not valid, for one of the following reasons:

- *ODDN* is completely blank (or blank up to the first null character in the field).
- 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.

This reason code can sometimes occur when a server application opens the reply queue specified by the *MDRQ* and *MDRM* fields in the MQMD of a message that the server has just received. In this case the reason code indicates that the application that sent the original message placed incorrect values into the *MDRQ* and *MDRM* fields in the MQMD of the original message.

Corrective action: Specify a valid name.

RC2012

(2012, X'7DC') Call not valid in environment.

The call is not valid for the current environment.

- On OS/390, one of the following applies:
 - An MQCONN call was issued, but the application had been linked with an adapter that is not supported in the environment in which the application is running. For example, this can arise when the application is linked with the MQ RRS adapter, but the application is running in a DB2 Stored Procedure address space. RRS is not supported in this environment. Stored Procedures wishing to use the MQ RRS adapter must run in a DB2 WLM-managed Stored Procedure address space.
 - An MQCMIT or MQBACK call was issued in the CICS or IMS environment.
- On Compaq (DIGITAL) OpenVMS, OS/2, OS/400, Tandem NonStop Kernel, 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.

Return codes

Corrective action: Do one of the following (as appropriate):

- On OS/390:
 - Link the application with the correct adapter.
 - For a CICS or IMS application, issue the appropriate CICS or IMS call to commit or backout the unit of work.
- In the other environments:
 - Link the application with the correct libraries (threaded or nonthreaded).
 - Remove from the application the call that is not supported.

RC2013

(2013, X'7DD') Expiry time not valid.

On an MQPUT or MQPUT1 call, the value specified for the *MDEXP* field in the message descriptor MQMD is not valid.

Corrective action: Specify a value which is greater than zero, or the special value EIULIM.

RC2014

(2014, X'7DE') Feedback code not valid.

On an MQPUT or MQPUT1 call, the value specified for the *MDFB* field in the message descriptor MQMD is not valid. The value is not FBNONE, and is outside both the range defined for system feedback codes and the range defined for application feedback codes.

Corrective action: Specify FBNONE, or a value in the range FBSFST through FBSLST, or FBAFST through FBALST.

RC2016

(2016, X'7E0') Gets inhibited for the queue.

MQGET calls are currently inhibited for the queue (see the *InhibitGet* queue attribute described in “Chapter 37. Attributes for all queues” on page 293), or for the queue to which this queue resolves (see “Chapter 40. Attributes for alias queues” on page 315).

Corrective action: If the system design allows get requests to be inhibited for short periods, retry the operation later.

RC2017

(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.

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 “Chapter 43. Attributes for the queue manager” on page 323).

RC2018

(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 parameter pointers that are not valid; 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 “Chapter 27. MQCONN - Connect queue manager” on page 221).

RC2019

(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 parameter pointers that are not valid; 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 “Chapter 32. MQOPEN - Open object” on page 251).

RC2020

(2020, X'7E4') Value for inhibit-get or inhibit-put queue attribute not valid.

On an MQSET call, the value specified for either the IAIGET attribute or the IAIPUT attribute is not valid.

Corrective action: Specify a valid value. See the *InhibitGet* or *InhibitPut* attribute described in “Chapter 37. Attributes for all queues” on page 293.

RC2021

(2021, X'7E5') Count of integer attributes not valid.

On an MQINQ or MQSET call, the *IACNT* parameter is negative (MQINQ or MQSET), or smaller than the number of integer attribute selectors (IA*) specified in the *SELS* parameter (MQSET only). This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Specify a value large enough for all selected integer attributes.

RC2022

(2022, X'7E6') Not enough space allowed for integer attributes.

On an MQINQ call, the *IACNT* parameter is smaller than the number of integer attribute selectors (IA*) specified in the *SELS* parameter.

The call completes with CCWARN, with the *INTATR* 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.

Return codes

RC2023

(2023, X'7E7') Integer attributes array not valid.

On an MQINQ or MQSET call, the *INTATR* 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 *IACNT* parameter (MQINQ only). (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2024

(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 PMSYP option
- Messages retrieved by the application with the GMSYP option
- Trigger messages and COA report messages generated by the queue manager for messages put with the PMSYP option
- COD report messages generated by the queue manager for messages retrieved with the GMSYP 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 OS/400, the limit for the maximum number of uncommitted messages can be changed by using the CHGMQM command.

RC2026

(2026, X'7EA') Message descriptor not valid.

MQMD structure is not valid. Either the *MDSID* mnemonic eye-catcher is not valid, or the *MDVER* is not recognized.

This reason also occurs if:

- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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.

RC2027

(2027, X'7EB') Missing reply-to queue.

On an MQPUT or MQPUT1 call, the *MDRQ* field in the message descriptor MQMD is blank, but one or both of the following is true:

- A reply was requested (that is, MTRQST was specified in the *MDMT* field of the message descriptor).
- A report message was requested in the *MDREP* 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.

RC2029

(2029, X'7ED') Message type in message descriptor not valid.

On an MQPUT or MQPUT1 call, the value specified for the *MDMT* field in the message descriptor (MQMD) is not valid.

Corrective action: Specify a valid value. See the *MDMT* field described in “Chapter 10. MQMD - Message descriptor” on page 83 for details.

RC2030

(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 MFSEGA was not specified in the *MDMFL* 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 MFSEGA 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 FMSTR (for FMSTR the minimum segment size is 16 bytes).

RC2030 can also occur in the *MDFB* 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 *BUFLN* 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 MFSEGA in the *MDMFL* field in MQMD; this will allow the queue manager to break the message into segments.

RC2031

(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 MFSEGA was not specified in the *MDMFL* 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.

Return codes

This reason code can also occur when *MFSEGA* 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 FMSTR (for FMSTR the minimum segment size is 16 bytes).

RC2031 can also occur in the *MDFB* 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.

Corrective action: Check whether the *BUFLEN* 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 *MFSEGA* in the *MDMFL* field in MQMD; this will allow the queue manager to break the message into segments.
- Check the channel definitions.

RC2033

(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 *MDMID* and *MDCID* fields), and in MQGMO (the *GMOPT* and *GMMO* fields). Either the GMWT option was not specified, or the time interval specified by the *GMWI* 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.

This reason code can also be returned by the mqGetBag and mqExecute calls. mqGetBag is similar to MQGET. For the mqExecute call, the completion code can be either MQCC_WARNING or MQCC_FAILED:

- If the completion code is MQCC_WARNING, some response messages were received during the specified wait interval, but not all. The response bag contains system-generated nested bags for the messages that were received.
- If the completion code is MQCC_FAILED, no response messages were received during the specified wait interval.

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:

GMLOGO
 GMAMSA
 GMASGA
 GMCMPM
 MOMSGI
 MOCORI
 MOGRPI
 MOSEQN
 MOOFFS
MDMID field
MDCID field

Consider waiting longer for the message.

RC2034

(2034, X'7F2') Browse cursor not positioned on message.

An MQGET call was issued with either the GMMUC or the GMBRWC 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 GMLK option with the preceding browse MQGET call.

RC2035

(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 an MQOPEN or MQPUT1 call, the user is not authorized to open the object for the option(s) specified.
- On an MQCLOSE call, the user is not authorized to delete the object, which is a permanent dynamic queue, and the *HOB*J 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 *MDFB* 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.

Return codes

RC2036

(2036, X'7F4') Queue not open for browse.

An MQGET call was issued with one of the following options:

GMBRWF
GMBRWN
GMBRWC
GMMUC

but the queue had not been opened for browse.

Corrective action: Specify OOBRW when the queue is opened.

RC2037

(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:

OOINPS
OOINPX
OOINPQ

RC2038

(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 OOINQ when the object is opened.

RC2039

(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 OOOOUT when the queue is opened.

RC2040

(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 OOSSET when the object is opened.

RC2041

(2041, X'7F9') Object definition changed since opened.

Since the *HOB*J handle used on this call was returned by the MQOPEN call, object definitions that affect this object have been changed. See "Chapter 32. MQOPEN - Open object" on page 251 for more information.

This reason does not occur if the object handle is specified in the *PMCT* field of the *PMO* 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.

RC2042

(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 *OPTS* 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”.

Corrective action: System design should specify whether an application is to wait and retry, or take other action.

RC2043

(2043, X'7FB') Object type not valid.

On the MQOPEN or MQPUT1 call, the *ODOT* field in the object descriptor MQOD specifies a value which is not valid. For the MQPUT1 call, the object type must be OTQ.

Corrective action: Specify a valid object type.

RC2044

(2044, X'7FC') Object descriptor structure not valid.

On the MQOPEN or MQPUT1 call, the object descriptor MQOD is not valid. Either the *ODSID* mnemonic eye-catcher is not valid, or the *ODVER* is not recognized.

This reason also occurs if:

- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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.

RC2045

(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 the following cases:

- An option that is inappropriate for the object type (for example, OOOUT for an OTPRO object).
- An option that is unsupported for the queue type (for example, OOINQ for a remote queue that has no local definition).
- One or more of the following options:
 - OOINPQ
 - OOINPS
 - OOINPX
 - OOBRW

Return codes

OOINQ
OOSET

when either:

- the queue name is resolved through a cell directory, or
- *ODMN* 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.

For the MQCLOSE call, this includes the following case:

- The CODEL or COPURG option when the queue is not a dynamic queue.

This reason code can also occur on the MQOPEN call when the object being opened is of type OTNLST, OTPRO, or OTQM, but the *ODMN* field in MQOD is neither blank nor the name of the local queue manager.

Corrective action: Specify the correct option; see Table 48 on page 256 for open options, and Table 47 on page 212 for close options. For the MQOPEN call, ensure that the *ODMN* field is set correctly. For the MQCLOSE call, either correct the option or change the definition type of the model queue that is used to create the new queue.

RC2046

(2046, X'7FE') Options not valid or not consistent.

The *OPTS* parameter or field contains options that are not valid, or a combination of options that is not valid.

- For the MQOPEN, MQCLOSE, MQXCNVC, mqBagToBuffer, mqBufferToBag, mqCreateBag, and mqExecute calls, *GMOPT* 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 parameter pointers that are not valid; if not detected, unpredictable results occur.)

- For the MQCONN, MQGET, MQPUT, and MQPUT1 calls, *GMOPT* is a field in the relevant options structure (MQGMO or MQPMO).

Corrective action: Specify valid options. Check the description of the *OPTS* 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.

RC2047

(2047, X'7FF') Persistence not valid.

On an MQPUT or MQPUT1 call, the value specified for the *MDPER* field in the message descriptor MQMD is not valid.

Corrective action: Specify one of the following values:

PEPER
PENPER
PEQDEF

RC2048

(2048, X'800') Message on a temporary dynamic queue cannot be persistent.

On an MQPUT or MQPUT1 call, the value specified for the *MDPER* field in the message descriptor MQMD specifies PEPER, 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 *MDFB* 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 PENPER 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 MTRPLY) with the same persistence as the original request message (message type MTRQST). If the request message is persistent, the reply queue specified in the *MDRQ* 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.

RC2049

(2049, X'801') Message Priority exceeds maximum value supported.

On an MQPUT or MQPUT1 call, the value of the *MDPRI* field in the message descriptor MQMD exceeds the maximum priority supported by the local queue manager (see the *MaxPriority* queue-manager attribute described in “Chapter 43. Attributes for the queue manager” on page 323). The message is accepted by the queue manager, but is placed on the queue at the queue manager’s maximum priority. The *MDPRI* field in the message descriptor retains the value specified by the application that put the message.

Corrective action: None required, unless this reason code was not expected by the application that put the message.

RC2050

(2050, X'802') Message priority not valid.

On an MQPUT or MQPUT1 call, the value of the *MDPRI* field in the message descriptor MQMD is not valid.

Corrective action: Specify a value which is zero or greater, or the special value PRQDEF.

RC2051

(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 “Chapter 37. Attributes for all queues” on page 293), or for the queue to which this queue resolves (see “Chapter 40. Attributes for alias queues” on page 315).

This reason code can also occur in the *MDFB* 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.

RC2052

(2052, X'804') Queue has been deleted.

Return codes

An *HOBJ* queue handle specified on a call refers to a dynamic queue that has been deleted since the queue was opened. (See “Chapter 25. MQCLOSE - Close object” on page 211 for information about the deletion of dynamic queues.)

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.

RC2053

(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 “Chapter 38. Attributes for local queues and model queues” on page 299).

This reason code can also occur in the *MDFB* 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.

RC2055

(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 CODEL option specified, but there are messages still on the queue, or
- The CODEL or COPURG 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 COPURG option. Consider retrying the call later.

RC2056

(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 *MDFB* 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 an application is putting messages in an infinite loop. If not, make more disk space available for the queue.

RC2057

(2057, X'809') Queue type not valid.

One of the following occurred:

- On an MQOPEN call, the *ODMN* 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 *ODON* 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 *MDRQ* 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 RC2057 is returned in the *DLREA* 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.

RC2058

(2058, X'80A') Queue manager name not valid or not known.

On an MQCONN call, the value specified for the *QMNAME* parameter is not valid. This reason also occurs if the parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

This reason also occurs if an application attempts to connect to a queue manager within a group (see the *QMNAME* 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.

RC2059

(2059, X'80B') Queue manager not available for connection.

On an MQCONN call, the queue manager identified by the *QMNAME* parameter is not available for connection at this time.

- On OS/400, this reason can also be returned by the MQOPEN and MQPUT1 calls, when HCDEFH is specified for the *HCONN* parameter by an application running in compatibility mode.

Return codes

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.

This reason also occurs if an application attempts to connect to a queue manager within a group (see the *QMNAME* 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.

RC2061

(2061, X'80D') Report options in message descriptor not valid.

An *MQPUT* or *MQPUT1* call was issued, but the *MDREP* 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 E. Report options" on page 457 for more details.

This reason code can also occur in the *MDFB* field in the *MQMD* of a report message, or in the *DLREA* 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 *MDREP* 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 *RONONE* if no report options are required.
2. Ensure that the report options specified are ones which are documented in this book; see the *MDREP* field described in "Chapter 10. *MQMD* - Message descriptor" on page 83 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 *ROEXC* and *ROEXCD* to the *MDREP* field; only one of these can be specified.

RC2063

(2063, X'80F') Security error occurred.

An *MQOPEN*, *MQPUT1*, or *MQCLOSE* call was issued, but it failed because a security error occurred.

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.

RC2065

(2065, X'811') Count of selectors not valid.

On an *MQINQ* or *MQSET* call, the *SELCNT* 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 parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Specify a value in the range 0 through 256.

RC2066

(2066, X'812') Count of selectors too big.

On an MQINQ or MQSET call, the *SELCNT* 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.

RC2067

(2067, X'813') Attribute selector not valid.

On an MQINQ or MQSET call, a selector in the *SELS* 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 parameter pointers that are not valid; 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.

RC2068

(2068, X'814') Selector not applicable to queue type.

On the MQINQ call, one or more selectors in the *SELS* array is not applicable to the type of the queue whose attributes are being inquired.

This reason also occurs when the queue is a cluster queue that resolved to a remote instance of the queue. In this case only a subset of the attributes that are valid for local queues can be inquired. See the usage notes in "Chapter 31. MQINQ - Inquire about object attributes" on page 241 for further details.

The call completes with CCWARN, with the attribute values for the inapplicable selectors set as follows:

- For integer attributes, the corresponding elements of *INTATR* are set to IAVNA.
- For character attributes, the appropriate parts of the *CHRATR* string are set to a character string consisting entirely of asterisks (*).

Corrective action: Verify that the selector specified is the one that was intended.

If the queue is a cluster queue, specifying one of the OOBRW, OOINP*, or OOSSET options in addition to OOINQ forces the queue to resolve to the local instance of the queue. However, if there is no local instance of the queue the MQOPEN call fails.

RC2071

(2071, X'817') Insufficient storage available.

The call failed because there is insufficient main storage available.

Return codes

Corrective action: Ensure that active applications are behaving correctly, for example, that they are not looping unexpectedly. If no problems are found, make more main storage available.

RC2072

(2072, X'818') Syncpoint support not available.

Either GMSYP was specified on an MQGET call or PMSYP 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 SPNAVL.

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 RC2072 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.

Corrective action: Remove the specification of GMSYP or PMSYP, 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.

RC2075

(2075, X'81B') Value for trigger-control attribute not valid.

On an MQSET call, the value specified for the IATRGC attribute selector is not valid.

Corrective action: Specify a valid value. See “Chapter 38. Attributes for local queues and model queues” on page 299.

RC2076

(2076, X'81C') Value for trigger-depth attribute not valid.

On an MQSET call, the value specified for the IATRGD attribute selector is not valid.

Corrective action: Specify a value which is greater than zero. See “Chapter 38. Attributes for local queues and model queues” on page 299.

RC2077

(2077, X'81D') Value for trigger-message-priority attribute not valid.

On an MQSET call, the value specified for the IATRGP attribute selector is not valid.

Corrective action: Specify a value in the range 0 through the value of *MaxPriority* queue-manager attribute. See “Chapter 38. Attributes for local queues and model queues” on page 299.

RC2078

(2078, X'81E') Value for trigger-type attribute not valid.

On an MQSET call, the value specified for the IATRGT attribute selector is not valid.

Corrective action: Specify a valid value. See “Chapter 38. Attributes for local queues and model queues” on page 299.

RC2079

(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 GMATM 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 *DATLEN* 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.

RC2080

(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 GMATM 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 GMBRWF was specified, the browse cursor is positioned logically before the highest-priority message on the queue.

The *DATLEN* 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 *DATLEN*, or specify GMATM if not all of the message data is required.

RC2082

(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.

This reason code can also occur when *BaseQName* is the name of a cluster queue that cannot be resolved successfully.

Corrective action: Correct the queue definitions.

RC2085

(2085, X'825') Unknown object name.

On an MQOPEN or MQPUT1 call, the *ODMN* 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 *ODON* field in the object descriptor is not recognized for the specified object type.

This reason code can also occur when the queue is a cluster queue that is hosted on a remote queue manager, but the local queue manager does not have a defined route to the remote queue manager.

Return codes

See also RC2052.

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.

RC2086

(2086, X'826') Unknown object queue manager.

On an MQOPEN or MQPUT1 call, the *ODMN* field in the object descriptor MQOD does not satisfy the naming rules for objects. For more information, see the *MQSeries Application Programming Guide* for

This reason also occurs if the *ODOT* field in the object descriptor has the value OTQM, and the *ODMN* 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.

RC2087

(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:

- *ODMN* is blank or the name of the local queue manager, *ODON* is the name of a local definition of a remote queue (or an alias to one), and one of the following is true:
 - *RemoteQMGrName* is blank or the name of the local queue manager. Note that this error occurs even if *XmitQName* is not blank.
 - *XmitQName* is blank, but there is no transmission queue defined with the name of *RemoteQMGrName*, and the *DefXmitQName* queue-manager attribute is blank.
 - *RemoteQMGrName* and *RemoteQName* specify a cluster queue that cannot be resolved successfully, and the *DefXmitQName* queue-manager attribute is blank.
- *ODMN* is the name of a local definition of a remote queue (containing a queue-manager alias definition), and one of the following is true:
 - *RemoteQName* is not blank.
 - *XmitQName* is blank, but there is no transmission queue defined with the name of *RemoteQMGrName*, and the *DefXmitQName* queue-manager attribute is blank.
- *ODMN* is not:
 - Blank
 - The name of the local queue manager
 - The name of a transmission queue
 - The name of a queue-manager alias definition (that is, a local definition of a remote queue with a blank *RemoteQName*)

but the *DefXmitQName* queue-manager attribute is blank.

- *ODMN* 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, and the *DefXmitQName* queue-manager attribute is blank.

Corrective action: Check the values specified for *ODMN* and *ODON*. If these are correct, check the queue definitions.

RC2090

(2090, X'82A') Wait interval in MQGMO not valid.

On the MQGET call, the value specified for the *GMWI* field in the *GMO* parameter is not valid.

Corrective action: Specify a value greater than or equal to zero, or the special value WIULIM if an indefinite wait is required.

RC2091

(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 *ODON* or *ODMN* 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 *ODON* and *ODMN*. If these are correct, check the queue definitions. For more information on transmission queues, see the *MQSeries Application Programming Guide*.

RC2092

(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:

- *ODMN* specifies the name of a local queue, but it does not have a *Usage* attribute of USTRAN.
- The *ODON* or *ODMN* 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 USTRAN
 - *XmitQName* is blank, but *RemoteQMgrName* specifies a queue that does not have a *Usage* attribute of USTRAN
- 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 USTRAN.

Corrective action: Check the values specified for *ODON* and *ODMN*. If these are correct, check the queue definitions. For more information on transmission queues, see the *MQSeries Application Programming Guide*.

RC2093

(2093, X'82D') Queue not open for pass all context.

An MQPUT call was issued with the PMPASA option specified in the *PMO* parameter, but the queue had not been opened with the OOPASA option.

Return codes

Corrective action: Specify OOPASA (or another option that implies it) when the queue is opened.

RC2094

(2094, X'82E') Queue not open for pass identity context.

An MQPUT call was issued with the PMPASI option specified in the *PMO* parameter, but the queue had not been opened with the OOPASI option.

Corrective action: Specify OOPASI (or another option that implies it) when the queue is opened.

RC2095

(2095, X'82F') Queue not open for set all context.

An MQPUT call was issued with the PMSETA option specified in the *PMO* parameter, but the queue had not been opened with the OOSETA option.

Corrective action: Specify OOSETA when the queue is opened.

RC2096

(2096, X'830') Queue not open for set identity context.

An MQPUT call was issued with the PMSETI option specified in the *PMO* parameter, but the queue had not been opened with the OOSETI option.

Corrective action: Specify OOSETI (or another option that implies it) when the queue is opened.

RC2097

(2097, X'831') Queue handle referred to does not save context.

On an MQPUT or MQPUT1 call, PMPASI or PMPASA was specified, but the handle specified in the *PMCT* field of the *PMO* parameter is either not a valid queue handle, or it is a valid queue handle but the queue was not opened with OOSAVA.

Corrective action: Specify OOSAVA when the queue referred to is opened.

RC2098

(2098, X'832') Context not available for queue handle referred to.

On an MQPUT or MQPUT1 call, PMPASI or PMPASA was specified, but the queue handle specified in the *PMCT* field of the *PMO* 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.

Corrective action: Ensure that a successful nonbrowse get call has been issued with the queue handle referred to.

RC2100

(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.

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.

RC2101

(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.

Corrective action: It may be necessary to stop and restart the queue manager, or to restore the queue-manager data from back-up storage.

Consult the FFST™ record to obtain more detail about the problem.

RC2102

(2102, X'836') Insufficient system resources available.

There are insufficient system resources to complete the call successfully.

Corrective action: Run the application when the machine is less heavily loaded.

Consult the FFST record to obtain more detail about the problem.

RC2104

(2104, X'838') Report option(s) in message descriptor not recognized.

An MQPUT or MQPUT1 call was issued, but the *MDREP* 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 E. Report options” on page 457 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 *MDREP* 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 *MDREP* field described in “Chapter 10. MQMD - Message descriptor” on page 83 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 ROEXC and ROEXCD to the *MDREP* field; only one of these can be specified.

RC2110

(2110, X'83E') Message format not valid.

On an MQGET call with the GMCONV option included in the *GMO* parameter, one or both of the *MDCSI* and *MDENC* fields in the message differs

Return codes

from the corresponding field in the *MSGDSC* 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 *MDFMT* field in the message cannot be found.
- The format name in the message is FMNONE.
- 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 *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set 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.

RC2111

(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 GMCONV option is included in the *GMO* parameter; the coded character-set identifier in error is the *MDCSI* field in the message being retrieved. In this case, the message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

This reason can also occur on the MQGET call when the message contains one or more of the data-related MQ header structures (MQCIH, MQDLH, MQIIH, MQRMH), and the *MDCSI* field in the message specifies a character set that does not have SBCS characters for the characters that are valid in queue names. MQ header structures containing such characters are not valid, and so the message is returned unconverted. The Unicode character set UCS-2 is an example of such a character set.

If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set 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 *SRCCSI* parameter. Either the *SRCCSI* parameter specifies a value which is not valid or not supported, or the *SRCCSI* parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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 *SRCCSI* 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.

RC2112

(2112, X'840') Source integer encoding not recognized.

On an MQGET call, with the GMCONV option included in the *GMO* parameter, the *MDENC* value in the message being retrieved specifies an integer encoding that is not recognized. The message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set and encoding fields always correctly describe the relevant message data.

This reason code can also occur on the MQXCNVC call, when the *OPTS* parameter contains an unsupported DCCS* value, or when DCCSUN 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.

RC2113

(2113, X'841') Packed-decimal encoding in message not recognized.

On an MQGET call with the GMCONV option included in the *GMO* parameter, the *MDENC* value in the message being retrieved specifies a decimal encoding that is not recognized. The message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set 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.

RC2114

(2114, X'842') Floating-point encoding in message not recognized.

On an MQGET call, with the GMCONV option included in the *GMO* parameter, the *MDENC* value in the message being retrieved specifies a floating-point encoding that is not recognized. The message data is

Return codes

returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set 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.

RC2115

(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 GMCONV option is included in the *GMO* parameter; the coded character-set identifier in error is the *MDCSI* field in the *MSGDSC* parameter. In this case, the message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

This reason can also occur on the MQGET call when the message contains one or more of the data-related MQ header structures (MQCIH, MQDLH, MQIIH, MQRMH), and the *MDCSI* field in the *MSGDSC* parameter specifies a character set that does not have SBCS characters for the characters that are valid in queue names. The Unicode character set UCS-2 is an example of such a character set.

This reason can also occur on the MQXCNVC call; the coded character-set identifier in error is the *TGTCSI* parameter. Either the *TGTCSI* parameter specifies a value which is not valid or not supported, or the *TGTCSI* parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Check the character-set identifier that was specified for the *MDCSI* field in the *MSGDSC* parameter on the MQGET call, or that was specified for the *SRCCSI* 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.

RC2116

(2116, X'844') Target integer encoding not recognized.

On an MQGET call with the GMCONV option included in the *GMO* parameter, the *MDENC* value in the *MSGDSC* parameter specifies an integer encoding that is not recognized. The message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message being retrieved, and the call completes with CCWARN.

This reason code can also occur on the MQXCNVC call, when the *OPTS* parameter contains an unsupported DCCT* value, or when DCCTUN 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.

RC2117

(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

On an MQGET call with the GMCONV option included in the *GMO* parameter, the *MDENC* value in the *MSGDSC* parameter specifies a decimal encoding that is not recognized. The message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

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.

RC2118

(2118, X'846') Floating-point encoding specified by receiver not recognized.

On an MQGET call with the GMCONV option included in the *GMO* parameter, the *MDENC* value in the *MSGDSC* parameter specifies a floating-point encoding that is not recognized. The message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

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.

RC2119

(2119, X'847') Message data not converted.

On an MQGET call with the GMCONV option included in the *GMO* parameter, an error occurred during conversion of the data in the message. The message data is returned unconverted, the values of the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to those of the message returned, and the call completes with CCWARN.

If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set 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 *MDFMT*, *MDCSI* and *MDENC* parameters that were specified when the message was put. Also check that these values, and the *MDCSI* and *MDENC* specified in the *MSGDSC* parameter on the MQGET call, are supported for

Return codes

queue-manager conversion. If the required conversion is not supported, conversion must be carried out by the application.

RC2120

(2120, X'848') Converted data too big for buffer.

On an MQGET call with the GMCONV option included in the *GMO* 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.

The message is returned unconverted, with the *CMPCOD* parameter of the MQGET call set to CCWARN. If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set and encoding fields always correctly describe the relevant message data.

This reason can also occur on the MQXCNVC call, when the *TGTBUF* parameter is too small to 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 *DATLEN* parameter; in the case of a DBCS string or mixed SBCS/DBCS string, this length may be *less than* the length of *TGTBUF*.

Corrective action: For the MQGET call, check that the exit is converting the message data correctly and setting the output length *DATLEN* 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.

RC2121

(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, OS/400, 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.

RC2122

(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, OS/400, 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.

RC2123

(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.

RC2124

(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.

RC2125

(2125, X'84D') Bridge started.

Return codes

The IMS bridge has been started.

Corrective action: None. This reason code is only used to identify the corresponding event message.

RC2126

(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.

RC2128

(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, OS/400, 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.

RC2134

(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 *BOSID* mnemonic eye-catcher is not BOSIDV.
- The *BOVER* field is not BOVER1.
- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the MQBO structure. Ensure that required input fields are set correctly.

RC2135

(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.

RC2136

(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 CCWARN in this case.
- The call failed for all of the destinations, but for differing reasons. The completion code is CCFAIL 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.

RC2137

(2137, X'859') Object not opened successfully.

A queue or other MQ object could not be opened successfully, for one of the following reasons:

- An MQCONN or MQCONNX call was issued, but the queue manager was unable to open an object that is used internally by the queue manager. As a result, processing cannot continue. The error log will contain the name of the object that could not be opened.
- 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 *RRREA* 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:

- If the error occurred on the MQCONN or MQCONNX call, ensure that the required objects exist by running the following command and then retrying the application:

```
STRMQM -c qmgr
```

where *qmgr* should be replaced by the name of the queue manager.

- If the error occurred on the MQPUT call, 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.

RC2139

(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:

Return codes

- The *CNSID* mnemonic eye-catcher is not *CNSIDV*.
- The *CNVER* field is not *CNVER1*.
- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Correct the definition of the *MQCNO* structure. Ensure that required input fields are set correctly.

RC2141

(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.

RC2142

(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.

RC2143

(2143, X'85F') Source length parameter not valid.

On the *MQXCNVC* call, the *SRCLN* 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 *SRCLN* parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

This reason code can also occur on the *MQGET* call when the *GMCONV* option is specified. In this case it indicates that the *RC2143* reason was returned by an *MQXCNVC* call issued by the data conversion exit.

Corrective action: Specify a length that is zero or greater.

If the reason code occurs on the *MQGET* call, check that the logic in the data-conversion exit is correct.

RC2144

(2144, X'860') Target length parameter not valid.

On the *MQXCNVC* call, the *TGTLEN* parameter is not valid for one of the following reasons:

- *TGTLEN* is less than zero.
- The *TGTLEN* parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)
- The DCCFIL option is specified, but the value of *TGTLEN* is such that the target buffer cannot be filled completely with valid characters. This can occur when *TGTCSI* is a pure DBCS character set (such as UCS-2), but *TGTLEN* specifies a length that is an odd number of bytes.

This reason code can also occur on the MQGET call when the GMCONV option is specified. In this case it indicates that the RC2144 reason was returned by an MQXCNVC call issued by the data conversion exit.

Corrective action: Specify a length that is zero or greater. If the DCCFIL option is specified, and *TGTCSI* is a pure DBCS character set, ensure that *TGTLEN* specifies a length that is a multiple of two.

If the reason code occurs on the MQGET call, check that the logic in the data-conversion exit is correct.

RC2145

(2145, X'861') Source buffer parameter not valid.

On the MQXCNVC call, the *SRCBUF* parameter pointer is not valid, or points to storage that cannot be accessed for the entire length specified by *SRCLEN*. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

This reason code can also occur on the MQGET call when the GMCONV option is specified. In this case it indicates that the RC2145 reason was returned by an MQXCNVC call issued by the data conversion exit.

Corrective action: Specify a valid buffer.

If the reason code occurs on the MQGET call, check that the logic in the data-conversion exit is correct.

RC2146

(2146, X'862') Target buffer parameter not valid.

On the MQXCNVC call, the *TGTBUF* parameter pointer is not valid, or points to read-only storage, or to storage that cannot be accessed for the entire length specified by *TGTLEN*. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

This reason code can also occur on the MQGET call when the GMCONV option is specified. In this case it indicates that the RC2146 reason was returned by an MQXCNVC call issued by the data conversion exit.

Corrective action: Specify a valid buffer.

If the reason code occurs on the MQGET call, check that the logic in the data-conversion exit is correct.

RC2148

(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.

Return codes

Corrective action: Correct the definition of the MQIIH structure. Ensure that the fields are set correctly.

RC2149

(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:

FMADMN
FMEVNT
FMPCF

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.

RC2150

(2150, X'866') DBCS string not valid.

On the MQXCNCV call, the *SRCCSI* parameter specifies the coded character-set identifier of a double-byte character set (DBCS), but the *SRCBUF* 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.

This reason code can also occur on the MQGET call when the GMCONV option is specified. In this case it indicates that the RC2150 reason was returned by an MQXCNCV call issued by the data conversion exit.

Corrective action: Specify a valid string.

If the reason code occurs on the MQGET call, check that the data in the message is valid, and that the logic in the data-conversion exit is correct.

RC2152

(2152, X'868') Object name not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *ODREC* field in MQOD is greater than zero), but the *ODON* field is neither blank nor the null string.

Corrective action: If it is intended to open a distribution list, set the *ODON* field to blanks or the null string. If it is not intended to open a distribution list, set the *ODREC* field to zero.

RC2153

(2153, X'869') Object queue-manager name not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *ODREC* field in MQOD is greater than zero), but the *ODMN* field is neither blank nor the null string.

Corrective action: If it is intended to open a distribution list, set the *ODMN* field to blanks or the null string. If it is not intended to open a distribution list, set the *ODREC* field to zero.

RC2154

(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:

- *ODREC* in MQOD is less than zero.
- *ODOT* in MQOD is not OTQ, and *ODREC* is not zero. *ODREC* must be zero if the object being opened is not a queue.

Corrective action: If it is intended to open a distribution list, set the *ODOT* field to OTQ and *ODREC* to the number of destinations in the list. If it is not intended to open a distribution list, set the *ODREC* field to zero.

RC2155

(2155, X'86B') Object records not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *ODREC* field in MQOD is greater than zero), but the MQOR object records are not specified correctly. One of the following applies:

- *ODORO* is zero and *ODORP* is the null pointer or zero.
- *ODORO* is not zero and *ODORP* is neither the null pointer nor zero.
- *ODORP* is not a valid pointer.
- *ODORP* or *ODORO* points to storage that is not accessible.

Corrective action: Ensure that one of *ODORO* and *ODORP* is zero and the other nonzero. Ensure that the field used points to accessible storage.

RC2156

(2156, X'86C') Response records not valid.

An MQOPEN or MQPUT1 call was issued to open a distribution list (that is, the *ODREC* field in MQOD is greater than zero), but the MQRR response records are not specified correctly. One of the following applies:

- *ODRRO* is not zero and *ODRRP* is neither the null pointer nor zero.
- *ODRRP* is not a valid pointer.
- *ODRRP* or *ODRRO* points to storage that is not accessible.

Corrective action: Ensure that at least one of *ODRRO* and *ODRRP* is zero. Ensure that the field used points to accessible storage.

RC2158

(2158, X'86E') Put message record flags not valid.

An MQPUT or MQPUT1 call was issued to put a message, but the *PMPRF* 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, *PMREC* is greater than zero, and one of *PMPRO* or *PMPRP* is nonzero), but *PMPRF* has the value PFNONE.
- PFACC is specified without either PMSETI or PMSETA.

Corrective action: Ensure that *PMPRF* is set with the appropriate PF* flags to indicate which fields are present in the put message records. If PFACC is specified, ensure that either PMSETI or PMSETA is also specified. Alternatively, set both *PMPRO* and *PMPRP* to zero.

RC2159

(2159, X'86F') Put message records not valid.

Return codes

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:

- *PMPRO* is not zero and *PMPRP* is neither the null pointer nor zero.
- *PMPRP* is not a valid pointer.
- *PMPRP* or *PMPRO* points to storage that is not accessible.

Corrective action: Ensure that at least one of *PMPRO* and *PMPRP* is zero. Ensure that the field used points to accessible storage.

RC2161

(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, 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 OOFIQ included in the *OPTS* parameter
- MQGET, with GMFIQ included in the *GMOPT* field of the *GMO* parameter
- MQPUT or MQPUT1, with PMFIQ included in the *PMOPT* field of the *PMO* parameter

Corrective action: The application should tidy up and stop. If the OOFIQ, PMFIQ, and GMFIQ 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.

RC2162

(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 GMWT option, the wait has been canceled. No more message-queuing calls can be issued.

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.

RC2173

(2173, X'87D') Put-message options structure not valid.

On an MQPUT or MQPUT1 call, the MQPMO structure is not valid. Either the *PMSID* mnemonic eye-catcher is not valid, or the *PMVER* is not recognized.

This reason also occurs if:

- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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.

RC2184

(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 *ODMN* field in the object descriptor was not blank and not the name of the local queue manager, but the *ODON* field is blank.

Corrective action: Alter the local definition of the remote queue and supply a valid remote queue name, or supply a nonblank *ODON* in the object descriptor, as appropriate.

RC2185

(2185, X'889') Inconsistent persistence specification.

The MQPUT call was issued to put a message that has a value for the *MDPER* field in MQMD that is different from the previous message put using that queue handle. This is not permitted when the PMLOGO 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.

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 *MDPER* field in MQMD.

RC2186

(2186, X'88A') Get-message options structure not valid.

On an MQGET call, the MQGMO structure is not valid. Either the *GMSID* mnemonic eye-catcher is not valid, or the *GMVER* is not recognized.

This reason also occurs if:

- The parameter pointer is not valid. (It is not always possible to detect parameter pointers that are not valid; 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.

RC2187

(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.

Return codes

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.

RC2188

(2188, X'88C') Call rejected by cluster workload exit.

An MQOPEN, MQPUT, or MQPUT1 call was issued to open or put a message on a cluster queue, but the cluster workload exit rejected the call.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Check the cluster workload exit to ensure that it has been written correctly. Determine why it rejected the call and correct the problem.

RC2189

(2189, X'88D') Cluster name resolution failed.

An MQOPEN, MQPUT, or MQPUT1 call was issued to open or put a message on a cluster queue, but the queue definition could not be resolved correctly because a response was required from the repository manager but none was available.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Check that the repository manager is operating and that the queue and channel definitions are correct.

RC2190

(2190, X'88E') Converted string too big for field.

On an MQGET call with the GMCONV option included in the *GMO* parameter, a string in a fixed-length field in the message expanded during data conversion and exceeded the size of the field. When this happens, the queue manager tries discarding trailing blank characters and characters following the first null character, in order to make the string fit, but in this case there were insufficient characters that could be discarded.

This reason code can also occur for messages with a format name of FMIMVS. When this happens, it indicates that the IMS variable string expanded such that its length exceeded the capacity of the 2-byte binary length field contained within the structure of the IMS variable string. (The queue manager never discards trailing blanks in an IMS variable string.)

The message is returned unconverted, with the *CMPCOD* parameter of the MQGET call set to CCWARN. If the message consists of several parts, each of which is described by its own character-set and encoding fields (for example, a message with format name FMDLH), some parts may be converted and other parts not converted. However, the values returned in the various character-set and encoding fields always correctly describe the relevant message data.

This reason code does not occur if the string could be made to fit by discarding trailing blank characters.

Corrective action: Check that the fields in the message contain the correct values, and that the character-set identifiers specified by the sender and receiver of the message are correct. If they are, the layout of the data in the message must be modified to increase the lengths of the field(s) so that there is sufficient space to allow the string(s) to expand when converted.

RC2191

(2191, X'88F') Character trigger message structure not valid.

On an MQPUT or MQPUT1 call, the character trigger message structure MQTMC in the message data is not valid.

Corrective action: Correct the definition of the MQTMC structure. Ensure that the fields are set correctly.

RC2194

(2194, X'892') Object name not valid for object type.

An MQOPEN call was issued to open the queue manager definition, but the *ODON* field in the *OBJDSC* parameter is not blank.

Corrective action: Ensure that the *ODON* field is set to blanks.

RC2195

(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.

Consult the FFST record to obtain more detail about the problem.

RC2196

(2196, X'894') Unknown transmission queue.

On an MQOPEN or MQPUT1 call, a message is to be sent to a remote queue manager. The *ODON* or the *ODMN* 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 *ODON* and *ODMN*. If these are correct, check the queue definitions. For more information on transmission queues, see the *MQSeries Application Programming Guide*

RC2197

(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.

RC2198

(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,

Return codes

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.

RC2199

(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 USTRAN.

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 USTRAN.

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

RC2206

(2206, X'89E') Message-identifier error.

On an MQGET call, the *MDMID* field is not zero. Selective retrieval by message identifier is not supported.

Corrective action: Ensure that the *MDMID* field is zero.

RC2207

(2207, X'89F') Correlation-identifier error.

On an MQGET call, the *MDCID* field is not zero. Selective retrieval by correlation identifier is not supported.

Corrective action: Ensure that the *MDCID* field is zero.

RC2208

(2208, X'8A0') File-system error.

An unexpected return was received from the file system, in attempting to perform an operation on a queue.

This reason code occurs only on VSE/ESA.

Corrective action: Check the file system definition for the queue that was being accessed. For a VSAM file, check that the control interval is large enough for the maximum message length allowed for the queue.

RC2209

(2209, X'8A1') No message locked.

An MQGET call was issued with the GMUNLK option, but no message was currently locked.

Corrective action: Check that a message was locked by an earlier MQGET call with the GMLK option for the same handle, and that no intervening call has caused the message to become unlocked.

RC2218

(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 *MDFB* field in the message descriptor of a report message.

Corrective action: Check the channel definitions. Increase the maximum message length that the channel can accept, or break the message into several smaller messages.

RC2219

(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.

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.

RC2220

(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.

Corrective action: Correct the definition of the MQRMH structure. Ensure that the fields are set correctly.

RC2222

(2222, X'8AE') Queue manager created.

This condition is detected when a queue manager becomes active.

Corrective action: None. This reason code is only used to identify the corresponding event message.

Return codes

RC2223

(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.

RC2224

(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.

RC2225

(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.

RC2226

(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.

RC2227

(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.

RC2232

(2232, X'8B8') Unit of work not started.

An MQGET, MQPUT or MQPUT1 call was issued to get or put a message within a unit of work, but no TM/MP transaction had been started. If GMNSYP is not specified on MQGET, or PMNSYP is not specified on MQPUT or MQPUT1 (the default), the call requires a unit of work.

Corrective action: Ensure a TM/MP transaction is available, or issue the MQGET call with the GMNSYP option, or the MQPUT or MQPUT1 call with the PMNSYP option, which will cause a transaction to be started automatically.

RC2233

(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.

Corrective action: None. This reason code is only used to identify the corresponding event message.

RC2234

(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.

Corrective action: Examine the additional information returned in the event message to determine the reason for the failure.

RC2235

(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.

RC2236

(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.

RC2237

(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.

RC2238

(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.

RC2239

(2239, X'8BF') PCF string parameter structure not valid.

Return codes

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.

RC2241

(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 PMLOGO. 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 PMLOGO, but the previous MQPUT call for the queue handle did specify PMLOGO. The call succeeds with completion code CCWARN 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 GMLOGO, but the previous MQGET call for the queue handle did specify GMLOGO. The call succeeds with completion code CCWARN 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 CCWARN.

If there is an incomplete logical message as well as an incomplete message group, reason code RC2242 is returned in preference to RC2241.

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 MFLMIG.

RC2242

(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 PMLOGO. 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 PMLOGO, but the previous MQPUT call for the queue handle did specify PMLOGO. The call succeeds with completion code CCWARN in this case.
- On the MQGET call, when the application attempts to get a message which is not the next segment, does *not* specify GMLOGO, but the previous MQGET call for the queue handle did specify GMLOGO. The call succeeds with completion code CCWARN 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 CCWARN.

Corrective action: If this reason code is expected, no corrective action is required. Otherwise, ensure that the MQPUT call for the last segment specifies MFLSEG.

RC2243

(2243, X'8C3') Message segments have differing CCSIDs.

An MQGET call was issued specifying the GMCMPM option, but the message to be retrieved consists of two or more segments which have differing values for the *MDCSI* 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 CCWARN, but only the first few segments that have identical character-set identifiers are returned.

Corrective action: Remove the GMCMPM option from the MQGET call and retrieve the remaining message segments one by one.

RC2244

(2244, X'8C4') Message segments have differing encodings.

An MQGET call was issued specifying the GMCMPM option, but the message to be retrieved consists of two or more segments which have differing values for the *MDENC* 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 CCWARN, but only the first few segments that have identical encodings are returned.

Corrective action: Remove the GMCMPM option from the MQGET call and retrieve the remaining message segments one by one.

RC2245

(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 PMSYP 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 PMLOGO, the call fails. If the current call does not specify PMLOGO, but the previous MQPUT call for the queue handle did, the call succeeds with completion code CCWARN.

- 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 GMSYP 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 GMLOGO, the call fails. If the current call does not specify GMLOGO, but the previous MQGET call for the queue handle did, the call succeeds with completion code CCWARN.

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.

RC2246

(2246, X'8C6') Message under cursor not valid for retrieval.

Return codes

An MQGET call was issued specifying the GMCMPM option with either GMMUC or GMBRWC, but the message that is under the cursor has an MQMD with an *MDOFF* field that is greater than zero. Because GMCMPM was specified, the message is not valid for retrieval.

Corrective action: Reposition the browse cursor so that it is located on a message whose *MDOFF* field in MQMD is zero. Alternatively, remove the GMCMPM option.

RC2247

(2247, X'8C7') Match options not valid.

An MQGET call was issued, but the value of the *GMMO* field in the *GMO* 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:

- GMLOGO is specified.
- There is a current message group or logical message for the queue handle.
- Neither of the following options is specified:
 - GMBRWC
 - GMMUC
- One or more of the MO* options is specified.
- The values of the fields in the *MSGDSC* parameter corresponding to the MO* options specified, differ from the values of those fields in the MQMD for the message to be returned next.

Corrective action: Ensure that only valid options are specified for the field.

RC2248

(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 *MESID* mnemonic eye-catcher is not MESIDV.
- The *MEVER* field is less than MEVER2.
- The *MELEN* field is less than MELEN2, or (for *MEVER* equal to MEVER2 only) greater than MELEN2.

Corrective action: Correct the definition of the message descriptor extension. Ensure that required input fields are correctly set.

RC2249

(2249, X'8C9') Message flags not valid.

An MQPUT or MQPUT1 call was issued, but the *MDMFL* 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 E. Report options" on page 457 for more details.

This reason code can also occur in the *MDFB* field in the MQMD of a report message, or in the *DLREA* 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.

Corrective action: Do the following:

1. Ensure that the *MDMFL* 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 MFNONE if no message flags are needed.
2. Ensure that the message flags specified are ones which are documented in this book; see the *MDMFL* field described in “Chapter 10. MQMD - Message descriptor” on page 83 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.

RC2250

(2250, X'8CA') Message sequence number not valid.

An MQGET, MQPUT, or MQPUT1 call was issued, but the value of the *MDSEQ* 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 *MDSEQ* field would have become greater than 999 999 999 as a result of the call.

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.

RC2251

(2251, X'8CB') Message segment offset not valid.

An MQPUT or MQPUT1 call was issued, but the value of the *MDOFF* 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 *MDOFF* field would have become greater than 999 999 999 as a result of the call.

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.

RC2252

(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 *MDOLN* 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)

Corrective action: Specify a value which is greater than zero. Zero is valid only for the last segment.

RC2253

(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.

Return codes

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.

RC2255

(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 GMCMPM 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 MFSEGA flag is specified in MQMD and the message requires segmentation.

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.

RC2256

(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 GMVER2, but the MQGMO supplied did not satisfy this condition.

Corrective action: Modify the application to pass a version-2 MQGMO. Check the application logic to ensure that the *GMVER* field in MQGMO has been set to GMVER2. Alternatively, remove the option that requires the version-2 MQGMO.

RC2257

(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 MDVER2, but the MQMD supplied did not satisfy this condition.

Corrective action: Modify the application to pass a version-2 MQMD. Check the application logic to ensure that the *MDVER* field in MQMD has been set to MDVER2. Alternatively, remove the option that requires the version-2 MQMD.

RC2258

(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:

- PMLOGO is not specified in the *PMOPT* field in MQPMO.
- Either there are too few MQPMR records provided by MQPMO, or the *PRGID* field is not present in the MQPMR records.

- One or more of the following flags is specified in the *MDMFL* field in MQMD or MQMDE:
MFSEGA
MF*MIG
MF*SEG
- The *MDGID* field in MQMD or MQMDE is not GINONE.

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.

Corrective action: Specify GINONE for the *MDGID* field in MQMD or MQMDE. Alternatively, if the call is MQPUT specify PMLOGO in the *PMOPT* field in MQPMO.

RC2259

(2259, X'8D3') Inconsistent browse specification.

An MQGET call was issued with the GMBRWN option specified, but the specification of the GMLOGO 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 GMLOGO, or neither call must specify GMLOGO.

Corrective action: Add or remove the GMLOGO option as appropriate. Alternatively, to switch between logical order and physical order, specify the GMBRWF option to restart the scan from the beginning of the queue, and either omit or specify GMLOGO as desired.

RC2260

(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.

Corrective action: Correct the definition of the MQXQH structure. Ensure that the fields are set correctly.

RC2261

(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:

- *RMSEL* is less than zero.
- *RMSEL* is greater than zero, but there is no source environment data.
- *RMSEL* is greater than zero, but *RMSEO* is negative, zero, or less than the length of the fixed part of MQRMH.
- *RMSEL* is greater than zero, but *RMSEO* plus *RMSEL* is greater than *RMLLEN*.

The exit returns this reason in the *CXFB* field of the MQCXP structure. If an exception report is requested, it is copied to the *CXFB* field of the MQMD associated with the report.

Corrective action: Specify the source environment data correctly.

RC2262

(2262, X'8D6') Source name data error.

Return codes

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:

- *RMSNL* is less than zero.
- *RMSNL* is greater than zero, but there is no source name data.
- *RMSNL* is greater than zero, but *RMSNO* is negative, zero, or less than the length of the fixed part of MQRMH.
- *RMSNL* is greater than zero, but *RMSNO* plus *RMSNL* is greater than *RMLEN*.

The exit returns this reason in the *CXFB* field of the MQCXP structure. If an exception report is requested, it is copied to the *CXFB* field of the MQMD associated with the report.

Corrective action: Specify the source name data correctly.

RC2263

(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:

- *RMDEL* is less than zero.
- *RMDEL* is greater than zero, but there is no destination environment data.
- *RMDEL* is greater than zero, but *RMDEO* is negative, zero, or less than the length of the fixed part of MQRMH.
- *RMDEL* is greater than zero, but *RMDEO* plus *RMDEL* is greater than *RMLEN*.

The exit returns this reason in the *CXFB* field of the MQCXP structure. If an exception report is requested, it is copied to the *CXFB* field of the MQMD associated with the report.

Corrective action: Specify the destination environment data correctly.

RC2264

(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:

- *MDNL* is less than zero.
- *MDNL* is greater than zero, but there is no destination name data.
- *MDNL* is greater than zero, but *MDNO* is negative, zero, or less than the length of the fixed part of MQRMH.
- *MDNL* is greater than zero, but *MDNO* plus *MDNL* is greater than *RMLEN*.

The exit returns this reason in the *CXFB* field of the MQCXP structure. If an exception report is requested, it is copied to the *CXFB* field of the MQMD associated with the report.

Corrective action: Specify the destination name data correctly.

RC2265

(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.

Corrective action: Correct the definition of the MQTM structure. Ensure that the fields are set correctly.

RC2266

(2266, X'8DA') Cluster workload exit failed.

An MQOPEN, MQPUT, or MQPUT1 call was issued to open or put a message on a cluster queue, but the cluster workload exit defined by the queue-manager's *ClusterWorkloadExit* attribute failed unexpectedly or did not respond in time. Subsequent MQOPEN, MQPUT, and MQPUT1 calls for this queue handle are processed as though the *ClusterWorkloadExit* attribute were blank.

- On OS/390, a message giving more information about the error is written to the system log, for example message CSQV455E or CSQV456E.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Check the cluster workload exit to ensure that it has been written correctly.

RC2267

(2267, X'8DB') Unable to load cluster workload exit.

An MQCONN call was issued to connect to a queue manager, but the call failed because the cluster workload exit defined by the queue-manager's *ClusterWorkloadExit* attribute could not be loaded.

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 cluster workload exit has been installed in the correct location.

RC2268

(2268, X'8DC') Put calls inhibited for all queues in cluster.

An MQOPEN call with the OOOOUT and OOBND0 options in effect was issued for a cluster queue, but all of the instances of the queue in the cluster are currently put-inhibited, that is, all of the queue instances have the *InhibitPut* attribute set to QAPUTI. Because there are no queue instances available to receive messages, the MQOPEN call fails.

Note: This reason code occurs only when both of the following are also true:

- There is no local instance of the queue. (If there is a local instance, the MQOPEN call succeeds, even if the local instance is put-inhibited.)
- There is no cluster workload exit for the queue, or there is a cluster workload exit but it did not choose a queue instance. (If the cluster workload exit does choose a queue instance, the MQOPEN call succeeds, even if that instance is put-inhibited.)

If the OOBNDN option is specified on the MQOPEN call, the call can succeed even if all of the queues in the cluster are put-inhibited. However, a subsequent MQPUT call may fail if all of the queues are still put-inhibited at the time of the MQPUT call.

Return codes

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: If the system design allows put requests to be inhibited for short periods, retry the operation later. If the problem persists, determine why all of the queues in the cluster are put-inhibited.

RC2269

(2269, X'8DD') Cluster resource error.

An MQOPEN, MQPUT, or MQPUT1 call was issued for a cluster queue, but an error occurred whilst trying to use a resource required for clustering.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Do the following:

- Check that the SYSTEM.CLUSTER.* queues are not put inhibited or full.
- Check the event queues for any events relating to the SYSTEM.CLUSTER.* queues, as these may give guidance as to the nature of the failure.
- Check that the repository queue manager is available.

RC2270

(2270, X'8DE') No destination queues available.

An MQPUT or MQPUT1 call was issued to put a message on a cluster queue, but at the time of the call there were no longer any instances of the queue in the cluster. The message therefore could not be sent.

This situation can occur when OOBNDN is specified on the MQOPEN call that opens the queue, or MQPUT1 is used to put the message.

This reason code occurs in the following environments: AIX, DOS client, HP-UX, OS/390, OS/2, OS/400, Sun Solaris, Windows client, Windows NT.

Corrective action: Check the queue definition and queue status to determine why all instances of the queue were removed from the cluster. Correct the problem and rerun the application.

RC2273

(2273, X'8E1') Error processing MQCONN call.

An MQCONN call failed for one of the following reasons:

- The system parameter module is not at the same release level as the queue manager.
- An internal error was detected by the queue manager.

This reason code occurs only on OS/390.

Corrective action: Relinkedit the system parameter module (CSQZPARM) to ensure that it is at the correct level. If the problem persists, contact your IBM support center.

RC2277

(2277, X'8E5') Channel definition not valid.

An MQCONN call was issued to connect to a queue manager, but the MQCD channel definition structure addressed by the CNCCO or CNCCP field

in MQCNO contains data that is not valid. Consult the MQSeries error log for more information about the nature of the error.

Corrective action: Ensure that required input fields in the MQCD structure are set correctly.

RC2278

(2278, X'8E6') Client connection fields not valid.

An MQCONN call was issued to connect to a queue manager, but the MQCD channel definition structure is not specified correctly. One of the following applies:

- *CNCCO* is not zero and *CNCCP* is neither the null pointer nor zero.
- *CNCCP* is not a valid pointer.
- *CNCCP* or *CNCCO* points to storage that is not accessible.

Corrective action: Ensure that at least one of *CNCCO* and *CNCCP* is zero. Ensure that the field used points to accessible storage.

RC2279

(2279, X'8E7') Channel stopped by user.

This condition is detected when the channel has been stopped by an operator. The reason qualifier identifies the reasons for stopping.

Corrective action: None. This reason code is only used to identify the corresponding event message.

RC2282

(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.

RC2283

(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.

RC2284

(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.

Return codes

RC2295

(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.

RC2296

(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. 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.

RC2299

(2299, X'8FB') Selector has wrong data type.

The *Selector* parameter has the wrong data type; it must be of type Long.

Corrective action: Declare the *Selector* parameter as Long.

RC2300

(2300, X'8FC') Command type not valid.

The mqExecute call was issued, but the value of the MQIASY_TYPE data item in the administration bag is not MQCFT_COMMAND.

Corrective action: Ensure that the MQIASY_TYPE data item in the administration bag has the value MQCFT_COMMAND.

RC2301

(2301, X'8FD') Multiple instances of system data item not valid.

The *Selector* parameter specifies a system selector (one of the MQIASY_* values), but the value of the *ItemIndex* parameter is not MQIND_NONE. Only one instance of each system selector can exist in the bag.

Corrective action: Specify MQIND_NONE for the *ItemIndex* parameter.

RC2302

(2302, X'8FE') System data item is read-only and cannot be altered.

A call was issued to modify the value of a system data item in a bag (a data item with one of the MQIASY_* selectors), but the call failed because the data item is one that cannot be altered by the application.

Corrective action: Specify the selector of a user-defined data item, or remove the call.

RC2303

(2303, X'8FF') Data could not be converted into a bag.

The `mqBufferToBag` or `mqGetBag` call was issued, but the data in the buffer or message could not be converted into a bag. This occurs when the data to be converted is not valid PCF.

Corrective action: Check the logic of the application that created the buffer or message to ensure that the buffer or message contains valid PCF.

If the message contains PCF that is not valid, the message cannot be retrieved using the `mqGetBag` call:

- If one of the GMBRW* options was specified, the message remains on the queue and can be retrieved using the `MQGET` call.
- In other cases, the message has already been removed from the queue and discarded. If the message was retrieved within a unit of work, the unit of work can be backed out and the message retrieved using the `MQGET` call.

RC2304

(2304, X'900') Selector not within valid range for call.

The *Selector* parameter has a value that is outside the valid range for the call. If the bag was created with the `MQCBO_CHECK_SELECTORS` option:

- For the `mqAddInteger` call, the value must be within the range `MQIA_FIRST` through `MQIA_LAST`.
- For the `mqAddString` call, the value must be within the range `MQCA_FIRST` through `MQCA_LAST`.

If the bag was not created with the `MQCBO_CHECK_SELECTORS` option:

- The value must be zero or greater.

Corrective action: Specify a valid value.

RC2305

(2305, X'901') Selector occurs more than once in bag.

The *ItemIndex* parameter has the value `MQIND_NONE`, but the bag contains more than one data item with the selector value specified by the *Selector* parameter. `MQIND_NONE` requires that the bag contain only one occurrence of the specified selector.

This reason code also occurs on the `mqExecute` call when the administration bag contains two or more occurrences of a selector for a required parameter that permits only one occurrence.

Corrective action: Check the logic of the application that created the bag. If correct, specify for *ItemIndex* a value that is zero or greater, and add application logic to process all of the occurrences of the selector in the bag.

Review the description of the administration command being issued, and ensure that all required parameters are defined correctly in the bag.

RC2306

(2306, X'902') Index not present.

The specified index is not present:

- For a bag, this means that the bag contains one or more data items that have the selector value specified by the *Selector* parameter, but none of them has the index value specified by the *ItemIndex* parameter. The data item identified by the *Selector* and *ItemIndex* parameters must exist in the bag.

Return codes

- For a namelist, this means that the index parameter value is too large, and outside the range of valid values.

Corrective action: Specify the index of a data item that does exist in the bag or namelist. Use the `mqCountItems` call to determine the number of data items with the specified selector that exist in the bag, or the `nameCount` method to determine the number of names in the namelist.

RC2307

(2307, X'903') String parameter not valid.

The *String* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2308

(2308, X'904') Encoding not supported.

The *MDENC* field in the message descriptor MQMD contains a value that is not supported:

- For the `mqPutBag` call, the field in error resides in the *MsgDesc* parameter of the call.
- For the `mqGetBag` call, the field in error resides in:
 - The *MsgDesc* parameter of the call if the GMCONV option was specified.
 - The message descriptor of the message about to be retrieved if GMCONV was *not* specified.

Corrective action: The value must be ENNAT.

If the value of the *MDENC* field in the message is not valid, the message cannot be retrieved using the `mqGetBag` call:

- If one of the GMBRW* options was specified, the message remains on the queue and can be retrieved using the MQGET call.
- In other cases, the message has already been removed from the queue and discarded. If the message was retrieved within a unit of work, the unit of work can be backed out and the message retrieved using the MQGET call.

RC2309

(2309, X'905') Selector not present in bag.

The *Selector* parameter specifies a selector that does not exist in the bag.

Corrective action: Specify a selector that does exist in the bag.

RC2310

(2310, X'906') OutSelector parameter not valid.

The *OutSelector* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2311

(2311, X'907') String truncated (too long for output buffer).

The string returned by the call is too long to fit in the buffer provided. The string has been truncated to fit in the buffer.

Corrective action: If the entire string is required, provide a larger buffer. On the `mqInquireString` call, the `StringLength` parameter is set by the call to indicate the size of the buffer required to accommodate the string without truncation.

RC2312

(2312, X'908') Selector implies a data type not valid for call.

A data item with the specified selector exists in the bag, but has a data type that conflicts with the data type implied by the call being used. For example, the data item might have an integer data type, but the call being used might be `mqSetString`, which implies a character data type.

This reason code also occurs on the `mqBagToBuffer`, `mqExecute`, and `mqPutBag` calls when `mqAddString` or `mqSetString` was used to add the `MQIACF_INQUIRY` data item to the bag.

Corrective action: For the `mqSetInteger` and `mqSetString` calls, specify `MQIND_ALL` for the `ItemIndex` parameter to delete from the bag all existing occurrences of the specified selector before creating the new occurrence with the required data type.

For the `mqInquireBag`, `mqInquireInteger`, and `mqInquireString` calls, use the `mqInquireItemInfo` call to determine the data type of the item with the specified selector, and then use the appropriate call to determine the value of the data item.

For the `mqBagToBuffer`, `mqExecute`, and `mqPutBag` calls, ensure that the `MQIACF_INQUIRY` data item is added to the bag using the `mqAddInteger` or `mqSetInteger` calls.

RC2313

(2313, X'909') Data type of item differs from previous occurrence of selector.

The `mqAddInteger` or `mqAddString` call was issued to add another occurrence of the specified selector to the bag, but the data type of this occurrence differed from the data type of the first occurrence.

This reason can also occur on the `mqBufferToBag` and `mqGetBag` calls, where it indicates that the PCF in the buffer or message contains a selector that occurs more than once but with inconsistent data types.

Corrective action: For the `mqAddInteger` and `mqAddString` calls, use the call appropriate to the data type of the first occurrence of that selector in the bag.

For the `mqBufferToBag` and `mqGetBag` calls, check the logic of the application that created the buffer or sent the message to ensure that multiple-occurrence selectors occur with only one data type. A message that contains a mixture of data types for a selector cannot be retrieved using the `mqGetBag` call:

- If one of the `GMBRW*` options was specified, the message remains on the queue and can be retrieved using the `MQGET` call.
- In other cases, the message has already been removed from the queue and discarded. If the message was retrieved within a unit of work, the unit of work can be backed out and the message retrieved using the `MQGET` call.

Return codes

RC2314

(2314, X'90A') Index not valid.

An index parameter to a call or method has a value that is not valid. The value must be zero or greater. For bag calls, certain MQIND_* values can also be specified:

- For the mqDeleteItem, mqSetInteger and mqSetString calls, MQIND_ALL and MQIND_NONE are valid.
- For the mqInquireBag, mqInquireInteger, mqInquireString, and mqInquireItemInfo calls, MQIND_NONE is valid.

Corrective action: Specify a valid value.

RC2315

(2315, X'90B') System bag is read-only and cannot be altered.

A call was issued to add a data item to a bag, modify the value of an existing data item in a bag, or retrieve a message into a bag, but the call failed because the bag is one that had been created by the system as a result of a previous mqExecute call. System bags cannot be modified by the application.

Corrective action: Specify the handle of a bag created by the application, or remove the call.

RC2316

(2316, X'90C') ItemCount parameter not valid.

The mqTruncateBag call was issued, but the *ItemCount* parameter specifies a value that is not valid. The value is either less than zero, or greater than the number of user-defined data items in the bag.

This reason also occurs on the mqCountItems call if the parameter pointer is not valid, or points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Specify a valid value. Use the mqCountItems call to determine the number of user-defined data items in the bag.

RC2317

(2317, X'90D') Format not supported.

The *MDFMT* field in the message descriptor MQMD contains a value that is not supported:

- For the mqPutBag call, the field in error resides in the *MsgDesc* parameter of the call.
- For the mqGetBag call, the field in error resides in the message descriptor of the message about to be retrieved.

Corrective action: The value must be one of the following:

FMADMN
FMEVNT
FMPCF

If the value of the *MDFMT* field in the message is none of these values, the message cannot be retrieved using the mqGetBag call:

- If one of the GMBRW* options was specified, the message remains on the queue and can be retrieved using the MQGET call.

- In other cases, the message has already been removed from the queue and discarded. If the message was retrieved within a unit of work, the unit of work can be backed out and the message retrieved using the MQGET call.

RC2318

(2318, X'90E') System selector not supported.

The *Selector* parameter specifies a value that is a system selector (a value that is negative), but the system selector is not one that is supported by the call.

Corrective action: Specify a selector value that is supported.

RC2319

(2319, X'90F') ItemValue parameter not valid.

The mqInquireBag or mqInquireInteger call was issued, but the *ItemValue* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2320

(2320, X'910') Bag handle not valid.

A call was issued that has a parameter that is a bag handle, but the handle is not valid. For output parameters, this reason also occurs if the parameter pointer is not valid, or points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2321

(2321, X'911') Parameter missing.

An administration message requires a parameter that is not present in the administration bag. This reason code occurs only for bags created with the MQCBO_ADMIN_BAG or MQCBO_REORDER_AS_REQUIRED options.

Corrective action: Review the description of the administration command being issued, and ensure that all required parameters are present in the bag.

RC2322

(2322, X'912') Command server not available.

The command server that processes administration commands is not available.

Corrective action: Start the command server.

RC2323

(2323, X'913') StringLength parameter not valid.

The *StringLength* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

Return codes

RC2324

(2324, X'914') Command code is not a recognized inquiry command.

The `mqAddInquiry` call was used previously to add attribute selectors to the bag, but the command code to be used for the `mqBagToBuffer`, `mqExecute`, or `mqPutBag` call is not recognized. As a result, the correct PCF message cannot be generated.

Corrective action: Remove the `mqAddInquiry` calls and use instead the `mqAddInteger` call with the appropriate `MQIACF_*_ATTRS` or `MQIACH_*_ATTRS` selectors.

RC2325

(2325, X'915') Input bag contains one or more nested bags.

A bag which is input to the call contains nested bags. Nested bags are supported only for bags which are output from the call.

Corrective action: Use a different bag as input to the call.

RC2326

(2326, X'916') Bag has wrong type for intended use.

The *Bag* parameter specifies the handle of a bag that has the wrong type for the call. The bag must be an administration bag, that is, it must be created with the `MQCBO_ADMIN_BAG` option specified on the `mqCreateBag` call.

Corrective action: Specify the `MQCBO_ADMIN_BAG` option when the bag is created.

RC2327

(2327, X'917') *ItemType* parameter not valid.

The `mqInquireItemInfo` call was issued, but the *ItemType* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2328

(2328, X'918') System bag is read-only and cannot be deleted.

An `mqDeleteBag` call was issued to delete a bag, but the call failed because the bag is one that had been created by the system as a result of a previous `mqExecute` call. System bags cannot be deleted by the application.

Corrective action: Specify the handle of a bag created by the application, or remove the call.

RC2329

(2329, X'919') System data item is read-only and cannot be deleted.

A call was issued to delete a system data item from a bag (a data item with one of the `MQIASY_*` selectors), but the call failed because the data item is one that cannot be deleted by the application.

Corrective action: Specify the selector of a user-defined data item, or remove the call.

RC2330

(2330, X'91A') Coded character set identifier parameter not valid.

The *CodedCharSetId* parameter is not valid. Either the parameter pointer is not valid, or it points to read-only storage. (It is not always possible to detect parameter pointers that are not valid; if not detected, unpredictable results occur.)

Corrective action: Correct the parameter.

RC2334

(2334, X'91E') MQRFH structure not valid.

The message contains an MQRFH structure, but the structure is not valid.

Corrective action: Modify the application that generated the message to ensure that it places a valid MQRFH structure in the message data.

RC2335

(2335, X'91F') *RFNVS* field not valid.

The contents of the *RFNVS* field in the MQRFH structure are not valid. *RFNVS* must adhere to the following rules:

- The string must consist of zero or more name/value pairs separated from each other by one or more blanks; the blanks are not significant.
- If a name or value contains blanks that are significant, the name or value must be enclosed in double-quote characters.
- If a name or value itself contains one or more double-quote characters, the name or value must be enclosed in double-quote characters, and each embedded double-quote character must be doubled.
- A name or value can contain any characters other than the null, which acts as a delimiter. The null and characters following it, up to the defined length of *RFNVS*, are ignored.

The following is a valid *RFNVS*:

```
Famous_Words "The program displayed ""Hello World"""
```

Corrective action: Modify the application that generated the message to ensure that it places in the *RFNVS* field data that adheres to the rules listed above. Check that the *RFLEN* field is set to the correct value.

RC2336

(2336, X'920') Command not valid.

The message contains an MQRFH structure, but the command name contained in the *RFNVS* field is not valid.

Corrective action: Modify the application that generated the message to ensure that it places in the *RFNVS* field a command name that is valid.

RC2337

(2337, X'921') Parameter not valid.

The message contains an MQRFH structure, but a parameter name contained in the *RFNVS* field is not valid for the command specified.

Corrective action: Modify the application that generated the message to ensure that it places in the *RFNVS* field only parameters that are valid for the specified command.

RC2338

(2338, X'922') Duplicate parameter.

The message contains an MQRFH structure, but a parameter occurs more than once in the *RFNVS* field when only one occurrence is valid for the specified command.

Return codes

Corrective action: Modify the application that generated the message to ensure that it places in the *RFNVS* field only one occurrence of the parameter.

RC2339

(2339, X'923') Parameter missing.

The message contains an MQRFH structure, but the command specified in the *RFNVS* field requires a parameter that is not present.

Corrective action: Modify the application that generated the message to ensure that it places in the *RFNVS* field all parameters that are required for the specified command.

Appendix B. MQSeries constants

Note

The names of the MQI constants are listed in this chapter in the form in which they appear in the RPG COPY file. The copy file is named CMQR.

This chapter specifies the values of all of the named constants that are used in the MQI, with the exception of those used for C++. For the C++ return codes refer to the *MQSeries Using C++* 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 "XX" that indicates the parameter or field to which the values relate. The constants are ordered alphabetically by this 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".
5. If the value is shown as "(variable)", it indicates that the value of the constant depends on the environment in which the application is running.

List of constants

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

LN* (Lengths of character string and byte fields)

See the *CHRATR* parameter described in "Chapter 31. MQINQ - Inquire about object attributes" on page 241 and "Chapter 35. MQSET - Set object attributes" on page 283.

LNABNC	4	X'00000004'
LNACCT	32	X'00000020'
LNAIDD	32	X'00000020'
LNAORD	4	X'00000004'
LNATID	4	X'00000004'
LNAUTH	8	X'00000008'
LNCID	24	X'00000018'
LNCLUN	48	X'00000030'
LNCNCL	4	X'00000004'
LNCRTD	12	X'0000000C'
LNCRTT	8	X'00000008'
LNDATE	12	X'0000000C'
LNEXN	20	X'00000014'
LNFAC	8	X'00000008'

MQSeries constants

LNFACL	4	X'00000004'
LNFM	8	X'00000008'
LNFUNC	4	X'00000004'
LNGLD	24	X'00000018'
LNLTOV	8	X'00000008'
LNMFMN	8	X'00000008'
LNMH	4000	X'0000FA0'
LNMI	24	X'00000018'
LNMTOK	16	X'00000010'
LNNLD	64	X'00000040'
LNNLN	48	X'00000030'
LNBJN	48	X'00000030'
LNOID	24	X'00000018'
LNPN	28	X'0000001C'
LNP	8	X'00000008'
LNPROA	256	X'0000100'
LNPROD	64	X'00000040'
LNPROE	128	X'00000080'
LNPRON	48	X'00000030'
LNPROU	128	X'00000080'
LNPTIM	8	X'00000008'
LNQD	64	X'00000040'
LNQMD	64	X'00000040'
LNQMID	48	X'00000030'
LNQMN	48	X'00000030'
LNQN	48	X'00000030'
LNRSID	4	X'00000004'
LNSTCO	4	X'00000004'
LNSTGC	8	X'00000008'
LNSVNM	32	X'00000020'
LNSVST	8	X'00000008'
LNTIID	16	X'00000010'
LNTIME	8	X'00000008'
LNTRGD	64	X'00000040'
LNTRID	4	X'00000004'
LNUI	12	X'0000000C'

AC* (Accounting token)

See the *MDACC* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

ACNONE X'00...00' (32 nulls)

ATT* (Accounting token type)

See the *MDACC* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

ATTUNK	X'00'
ATTCIC	X'01'
ATTOS2	X'04'
ATTDOS	X'05'
ATTUNX	X'06'
ATT400	X'08'
ATTWIN	X'09'
ATTWNT	X'0B'
ATTUSR	X'19'

AT* (Application type)

See the *MDPAT* field described in “Chapter 10. MQMD - Message descriptor” on page 83, and the *AppType* attribute described in “Chapter 42. Attributes for process definitions” on page 319.

ATUNK	-1	X'FFFFFFFF'
ATNCON	0	X'00000000'
ATCICS	1	X'00000001'
ATMVS	2	X'00000002'
AT390	2	X'00000002'
ATIMS	3	X'00000003'
ATOS2	4	X'00000004'
ATDOS	5	X'00000005'
ATAIX	6	X'00000006'
ATUNIX	6	X'00000006'
ATQM	7	X'00000007'
AT400	8	X'00000008'
ATDEF	8	X'00000008'
ATWIN	9	X'00000009'
ATVSE	10	X'0000000A'
ATWINT	11	X'0000000B'
ATVMS	12	X'0000000C'
ATGUAR	13	X'0000000D'
ATNSK	13	X'0000000D'
ATVOS	14	X'0000000E'
ATIMSB	19	X'00000013'
ATXCF	20	X'00000014'
ATCICB	21	X'00000015'
ATNOTE	22	X'00000016'
ATBRKR	26	X'0000001A'
ATJAVA	28	X'0000001C'
ATUFST	65536	X'00010000'
ATULST	999999999	X'3B9AC9FF'

BND* (Binding)

See the *DefBind* attribute described in “Chapter 37. Attributes for all queues” on page 293.

BNDOPN	0	X'00000000'
BNDNOT	1	X'00000001'

BO* (Begin options)

See the *BOOPT* field described in “Chapter 3. MQBO - Begin options” on page 15.

BONONE	0	X'00000000'
--------	---	-------------

BO* (Begin options structure identifier)

See the *BOSID* field described in “Chapter 3. MQBO - Begin options” on page 15.

BOSIDV	'B0bb'
--------	--------

BO* (Begin options version)

See the *BOVER* field described in “Chapter 3. MQBO - Begin options” on page 15.

MQSeries constants

BOVER1	1	X'00000001'
BOVERC	1	X'00000001'

CA* (Character attribute selector)

See the *SELS* parameter described in “Chapter 31. MQINQ - Inquire about object attributes” on page 241 and “Chapter 35. MQSET - Set object attributes” on page 283.

CAFRST	2001	X'000007D1'
CAAPPI	2001	X'000007D1'
CABASQ	2002	X'000007D2'
CACMDQ	2003	X'000007D3'
CACRTD	2004	X'000007D4'
CACRTT	2005	X'000007D5'
CADLQ	2006	X'000007D6'
CAENV D	2007	X'000007D7'
CAINI Q	2008	X'000007D8'
CALSTD	2009	X'000007D9'
CALSTN	2010	X'000007DA'
CAPROD	2011	X'000007DB'
CAPRON	2012	X'000007DC'
CAQD	2013	X'000007DD'
CAQMD	2014	X'000007DE'
CAQMN	2015	X'000007DF'
CAQN	2016	X'000007E0'
CARQMN	2017	X'000007E1'
CARQN	2018	X'000007E2'
CABRQN	2019	X'000007E3'
CANAMS	2020	X'000007E4'
CAUSR D	2021	X'000007E5'
CASTGC	2022	X'000007E6'
CATRGD	2023	X'000007E7'
CAXQN	2024	X'000007E8'
CADXQN	2025	X'000007E9'
CACADX	2026	X'000007EA'
CAALTD	2027	X'000007EB'
CAALTT	2028	X'000007EC'
CACLN	2029	X'000007ED'
CACLNL	2030	X'000007EE'
CACLQM	2031	X'000007EF'
CAQMID	2032	X'000007F0'
CACLWX	2033	X'000007F1'
CACLWD	2034	X'000007F2'
CARPN	2035	X'000007F3'
CARPNL	2036	X'000007F4'
CACLD	2037	X'000007F5'
CACLT	2038	X'000007F6'
CAUSER	4000	X'00000FA0'
CALAST	4000	X'00000FA0'
CALSTU	(variable)	

AD* (CICS header ADS descriptor)

See the *CIADS* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

ADNONE	0	X'00000000'
--------	---	-------------

ADSEND	1	X'00000001'
ADRECV	16	X'00000010'
ADMSGF	256	X'00000100'

CC* (Completion code)

See the *CMPCOD* parameter described in each MQI call.

CCOK	0	X'00000000'
CCWARN	1	X'00000001'
CCFAIL	2	X'00000002'

CS* (Coded character set identifier)

See the *MDCSI* field described in “Chapter 10. MQMD - Message descriptor” on page 83 and in other structures.

CT* (CICS header conversational task)

See the *CICT* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CTNO	0	X'00000000'
CTYES	1	X'00000001'

FC* (CICS header facility)

See the *CIFAC* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

FCNONE	X'00...00' (8 nulls)
--------	----------------------

CF* (CICS header function name)

See the *CIFNC* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CFCONN	'CONN'
CFGET	'GETb'
CFINQ	'INQb'
CFOPEN	'OPEN'
CFPUT	'PUTb'
CFPUT1	'PUT1'
CFNONE	'bbbb'

WI* (CICS header get-wait interval)

See the *CIGWI* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

WIDFLT	-2	X'FFFFFFFE'
--------	----	-------------

CI* (Correlation identifier)

See the *MDCID* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MQSeries constants

CINONE	X'00...00' (24 nulls)
CINEWS	X'414D51214E45575F534553...'

MQ* (Call identifier)

MQCONN	1	X'00000001'
MQDISC	2	X'00000002'
MQOPEN	3	X'00000003'
MQCLOS	4	X'00000004'
MQGET	5	X'00000005'
MQPUT	6	X'00000006'
MQPUT1	7	X'00000007'
MQINQ	8	X'00000008'
MQSET	9	X'00000009'
MQXCVC	12	X'0000000C'

CIF* (CICS header flags)

See the *CIFLG* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CIFNON	0	X'00000000'
--------	---	-------------

CI* (CICS header length)

See the *CILEN* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CILEN1	164	X'000000A4'
CILEN2	180	X'000000B4'
CILENC	180	X'000000B4'

CI* (CICS header structure identifier)

See the *CISID* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CISIDV	'CIHb'
--------	--------

CI* (CICS header version)

See the *CIVER* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CIVER1	1	X'00000001'
CIVER2	2	X'00000002'
CIVERC	2	X'00000002'

LT* (CICS header link type)

See the *CILT* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

LTPROG	1	X'00000001'
LTTRAN	2	X'00000002'

CMLV* (Command level)

See the *CommandLevel* attribute described in “Chapter 43. Attributes for the queue manager” on page 323.

CMLVL1	100	X'00000064'
CML320	320	X'00000140'
CML420	420	X'000001A4'
CML510	510	X'000001FE'

CN* (Connect options)

See the *CNOPT* field described in “Chapter 5. MQCNO - Connect options” on page 31.

CNSBND	0	X'00000000'
CNFBND	1	X'00000001'
CNNONE	0	X'00000000'

CN* (Connect options structure identifier)

See the *CNSID* field described in “Chapter 5. MQCNO - Connect options” on page 31.

CNSIDV	'CNOb'
--------	--------

CN* (Connect options version)

See the *CNVER* field described in “Chapter 5. MQCNO - Connect options” on page 31.

CNVER1	1	X'00000001'
CNVER2	2	X'00000002'
CNVERC	2	X'00000002'

CO* (Close options)

See the *OPTS* parameter described in “Chapter 25. MQCLOSE - Close object” on page 211.

CONONE	0	X'00000000'
CODEL	1	X'00000001'
COPURG	2	X'00000002'

OL* (CICS header output data length)

See the *CIODL* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

OLINPT	-1	X'FFFFFFFF'
--------	----	-------------

CRC* (CICS header return code)

See the *CIRET* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CRC000	0	X'00000000'
CRC001	1	X'00000001'

MQSeries constants

CRC002	2	X'00000002'
CRC003	3	X'00000003'
CRC004	4	X'00000004'
CRC005	5	X'00000005'
CRC006	6	X'00000006'
CRC007	7	X'00000007'
CRC008	8	X'00000008'
CRC009	9	X'00000009'

SC* (CICS header transaction start code)

See the *CISC* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

SCSTRT	'Sbbb'
SCDATA	'SDbb'
SCTERM	'TDbb'
SCNONE	'bbbb'

TE* (CICS header task end status)

See the *CITES* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

TENOSY	0	X'00000000'
TECMIT	256	X'00000100'
TEBACK	4352	X'00001100'
TEENDT	65536	X'00010000'

CU* (CICS header unit-of-work control)

See the *CIUOW* field described in “Chapter 4. MQCIH - CICS bridge header” on page 17.

CUMIDL	16	X'00000010'
CUFRST	17	X'00000011'
CUCMIT	256	X'00000100'
CULAST	272	X'00000110'
CUONLY	273	X'00000111'
CUBACK	4352	X'00001100'
CUCONT	65536	X'00010000'

DCC* (Convert-characters masks and factors)

See the *OPTS* parameter described in “MQXCNVC - Convert characters” on page 473.

DCCSMA	240	X'000000F0'
DCCTMA	3840	X'000000F00'
DCCSFA	16	X'00000010'
DCCTFA	256	X'00000100'

DCC* (Convert-characters options)

See the *OPTS* parameter described in “MQXCNVC - Convert characters” on page 473.

MQSeries constants

DCCSUN	0	X'00000000'
DCCTUN	0	X'00000000'
DCCNON	0	X'00000000'
DCCDEF	1	X'00000001'
DCCFIL	2	X'00000002'
DCCSNA	16	X'00000010'
DCCSNO	16	X'00000010'
DCCSRE	32	X'00000020'
DCCTNA	256	X'00000100'
DCCTNO	256	X'00000100'
DCCTRE	512	X'00000200'

DH* (Distribution header structure identifier)

See the *DHSID* field described in “Chapter 6. MQDH - Distribution header” on page 37.

DHSIDV 'DHbb'

DH* (Distribution header version)

See the *DHVER* field described in “Chapter 6. MQDH - Distribution header” on page 37.

DHVER1	1	X'00000001'
DHVERC	1	X'00000001'

DHF* (Distribution header flags)

See the *DHFLG* field described in “Chapter 6. MQDH - Distribution header” on page 37.

DHFNON	0	X'00000000'
DHFNEW	1	X'00000001'

DL* (Distribution list support)

See the *DistLists* attributes described in “Chapter 43. Attributes for the queue manager” on page 323 and “Chapter 38. Attributes for local queues and model queues” on page 299.

DLNSUP	0	X'00000000'
DLSUPP	1	X'00000001'

DL* (Dead-letter header structure identifier)

See the *DLSID* field described in “Chapter 7. MQDLH - Dead-letter header” on page 43.

DLSIDV 'DLHb'

DL* (Dead-letter header version)

See the *DLVER* field described in “Chapter 7. MQDLH - Dead-letter header” on page 43.

DLVER1	1	X'00000001'
--------	---	-------------

MQSeries constants

DLVERC	1	X'00000001'
--------	---	-------------

DX* (Data-conversion-exit parameter structure identifier)

See the *DXSID* field described in “MQDXP – Data-conversion exit parameter” on page 468.

DXSIDV	'DXPb'
--------	--------

DX* (Data-conversion-exit parameter structure version)

See the *DXVER* field described in “MQDXP – Data-conversion exit parameter” on page 468.

DXVER1	1	X'00000001'
DXVERC	1	X'00000001'

EI* (Expiry interval)

See the *MDEXP* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

EIULIM	-1	X'FFFFFFFF'
--------	----	-------------

EN* (Encoding)

See the *MDENC* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

ENNAT	273	X'00000111'
-------	-----	-------------

EN* (Encoding masks)

See “Appendix D. Machine encodings” on page 453.

ENIMSK	15	X'0000000F'
ENDMSK	240	X'000000F0'
ENFMSK	3840	X'00000F00'
ENRMSK	-4096	X'FFFFFF00'

EN* (Encoding for packed-decimal integers)

See “Appendix D. Machine encodings” on page 453.

ENDUND	0	X'00000000'
ENDNOR	16	X'00000010'
ENDREV	32	X'00000020'

EN* (Encoding for floating-point numbers)

See “Appendix D. Machine encodings” on page 453.

ENFUND	0	X'00000000'
ENFNOR	256	X'00000100'
ENFREV	512	X'00000200'
ENF390	768	X'00000300'

EN* (Encoding for binary integers)

See “Appendix D. Machine encodings” on page 453.

ENIUND	0	X'00000000'
ENINOR	1	X'00000001'
ENIREV	2	X'00000002'

EV* (Event reporting)

See the *QDepthHighEvent*, *QDepthLowEvent*, and *QDepthMaxEvent* attributes described in “Chapter 38. Attributes for local queues and model queues” on page 299, and the *AuthorityEvent*, *ChannelAutoDefEvent*, *InhibitEvent*, *LocalEvent*, *PerformanceEvent*, *RemoteEvent*, and *StartStopEvent* attributes described in “Chapter 43. Attributes for the queue manager” on page 323.

EVRDIS	0	X'00000000'
EVRENA	1	X'00000001'

FB* (Feedback)

See the *MDFB* field described in “Chapter 10. MQMD - Message descriptor” on page 83, and the *DLREA* field described in “Chapter 7. MQDLH - Dead-letter header” on page 43; see also the RC* values.

FBNONE	0	X'00000000'
FBSFST	1	X'00000001'
FBQUIT	256	X'00000100'
FBEXP	258	X'00000102'
FBCOA	259	X'00000103'
FBCOD	260	X'00000104'
FBABEG	265	X'00000109'
FBTM	266	X'0000010A'
FBATYP	267	X'0000010B'
FBSBMX	268	X'0000010C'
FBXQME	271	X'0000010F'
FBPAN	275	X'00000113'
FBNAN	276	X'00000114'
FBBCX	277	X'00000115'
FBSPS	279	X'00000117'
FBNARM	280	X'00000118'
FBDLZ	291	X'00000123'
FBDLN	292	X'00000124'
FBDLTB	293	X'00000125'
FBBUFO	294	X'00000126'
FBLOB1	295	X'00000127'
FBIH	296	X'00000128'
FBNAFI	298	X'0000012A'
FBIERR	300	X'0000012C'
FBIFST	301	X'0000012D'
FBILST	399	X'0000018F'
FBCINE	401	X'00000191'
FBCNTA	402	X'00000192'
FBCBRF	403	X'00000193'
FBCCIE	404	X'00000194'
FBCCSE	405	X'00000195'
FBCENE	406	X'00000196'
FBCIHE	407	X'00000197'
FBCUWE	408	X'00000198'

MQSeries constants

FBCCAE	409	X'00000199'
FBCANS	410	X'0000019A'
FBCAAB	411	X'0000019B'
FBCDLQ	412	X'0000019C'
FBCUBO	413	X'0000019D'
FBSLST	65535	X'0000FFFF'
FBAFST	65536	X'00010000'
FBALST	999999999	X'3B9AC9FF'

FM* (Format)

See the *MDFMT* field described in “Chapter 10. MQMD - Message descriptor” on page 83 and in other structures.

FMNONE	'bbbbbbbb'
FMADMN	'MQADMINb'
FMCICS	'MQCICSbb'
FMCMD1	'MQCMD1bb'
FMCMD2	'MQCMD2bb'
FMDLH	'MQDEADbb'
FMDH	'MQHDISTb'
FMEVNT	'MQEVENTb'
FMIMS	'MQIMSbbb'
FMIMVS	'MQIMSVSb'
FMMDE	'MQHMDEbb'
FMPCF	'MQPCFbbb'
FMRMH	'MQHREFbb'
FMRFH	'MQHRFbbb'
FMSTR	'MQSTRbbb'
FMTM	'MQTRIGbb'
FMWIH	'MQHWIHbb'
FMXQH	'MQXMITbb'

GI* (Group identifier)

See the *MDGID* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

GINONE	X'00...00' (24 nulls)
--------	-----------------------

GM* (Get message options)

See the *GMOPT* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

GMNWT	0	X'00000000'
GMNONE	0	X'00000000'
GMWT	1	X'00000001'
GMSYP	2	X'00000002'
GMNSYP	4	X'00000004'
GMBRWF	16	X'00000010'
GMBRWN	32	X'00000020'
GMATM	64	X'00000040'
GMMUC	256	X'00000100'
GMLK	512	X'00000200'
GMUNLK	1024	X'00000400'
GMBRWC	2048	X'00000800'
GMPSYP	4096	X'00001000'

GMFIQ	8192	X'00002000'
GMCONV	16384	X'00004000'
GMLOGO	32768	X'00008000'
GMCMPM	65536	X'00010000'
GMAMSA	131072	X'00020000'
GMASGA	262144	X'00040000'

GM* (Get message options structure identifier)

See the *GMSID* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

GMSIDV	'GMOb'
--------	--------

GM* (Get message options version)

See the *GMVER* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

GMVER1	1	X'00000001'
GMVER2	2	X'00000002'
GMVER3	3	X'00000003'
GMVERC	(variable)	

GS* (Group status)

See the *GMGST* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

GSNIG	'b'
GSMIG	'g'
GSLMIG	'L'

HC* (Connection handle)

See the *HCONN* parameter described in “Chapter 27. MQCONN - Connect queue manager” on page 221 and “Chapter 29. MQDISC - Disconnect queue manager” on page 229.

HCUNUH	-1	X'FFFFFFFF'
HCDEFH	0	X'00000000'

HO* (Object handle)

See the *HOBJ* parameter described in “Chapter 25. MQCLOSE - Close object” on page 211.

IA* (Integer attribute selector)

See the *SELS* parameter described in “Chapter 31. MQINQ - Inquire about object attributes” on page 241 and “Chapter 35. MQSET - Set object attributes” on page 283.

IAFRST	1	X'00000001'
IAAPPT	1	X'00000001'
IACCSI	2	X'00000002'
IACDEP	3	X'00000003'

MQSeries constants

IADINP	4	X'00000004'
IADPER	5	X'00000005'
IADPRI	6	X'00000006'
IADEFT	7	X'00000007'
IAHGB	8	X'00000008'
IAIGET	9	X'00000009'
IAIPUT	10	X'0000000A'
IAMHND	11	X'0000000B'
IAUSAG	12	X'0000000C'
IAMLEN	13	X'0000000D'
IAMPRI	14	X'0000000E'
IAMDEP	15	X'0000000F'
IAMDS	16	X'00000010'
IAOIC	17	X'00000011'
IAOOC	18	X'00000012'
IANAMC	19	X'00000013'
IAQTYP	20	X'00000014'
IARINT	21	X'00000015'
IABTHR	22	X'00000016'
IASHAR	23	X'00000017'
IATRGC	24	X'00000018'
IATRGI	25	X'00000019'
IATRGP	26	X'0000001A'
IATRGT	28	X'0000001C'
IATRGD	29	X'0000001D'
IASYNC	30	X'0000001E'
IACMDL	31	X'0000001F'
IAPLAT	32	X'00000020'
IAMUNC	33	X'00000021'
IADIST	34	X'00000022'
IATSR	35	X'00000023'
IAHQD	36	X'00000024'
IAMEC	37	X'00000025'
IAMDC	38	X'00000026'
IAQDHL	40	X'00000028'
IAQDLL	41	X'00000029'
IAQDME	42	X'0000002A'
IAQDHE	43	X'0000002B'
IAQDLE	44	X'0000002C'
IASCOP	45	X'0000002D'
IAQSIE	46	X'0000002E'
IAAUTE	47	X'0000002F'
IAINHE	48	X'00000030'
IACLE	49	X'00000031'
IARMTE	50	X'00000032'
IASSE	52	X'00000034'
IAPFME	53	X'00000035'
IAQSI	54	X'00000036'
IACAD	55	X'00000037'
IACADE	56	X'00000038'
IAINDT	57	X'00000039'
IACLWL	58	X'0000003A'
IACLQT	59	X'0000003B'
IAARCH	60	X'0000003C'
IADBND	61	X'0000003D'
IAUSER	2000	X'000007D0'
IALAST	2000	X'000007D0'
IALSTU	(variable)	

IAU* (IMS authenticator)

See the *IIAUT* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

IAUNON	'bbbbbbbb'
--------	------------

IAV* (Integer attribute value)

See the *INTATR* parameter described in “Chapter 31. MQINQ - Inquire about object attributes” on page 241.

IAVUND	-2	X'FFFFFFFE'
IAVNA	-1	X'FFFFFFF'

ICM* (IMS commit mode)

See the *IICMT* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

ICMCTS	'0'
ICMSTC	'1'

II* (IMS header flags)

See the *IIFLG* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

IINONE	0	X'00000000'
--------	---	-------------

II* (IMS header length)

See the *IILEN* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

IILEN1	84	X'00000054'
--------	----	-------------

II* (IMS header structure identifier)

See the *IISID* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

IISIDV	'IIHb'
--------	--------

II* (IMS header version)

See the *IIVER* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

IIVER1	1	X'00000001'
IIVERC	1	X'00000001'

ISS* (IMS security scope)

See the *IISEC* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

MQSeries constants

ISSCHK	'C'
ISSFUL	'F'

ITI* (IMS transaction instance identifier)

See the *IITID* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

ITINON	X'00...00' (16 nulls)
--------	-----------------------

ITS* (IMS transaction state)

See the *IITST* field described in “Chapter 9. MQIIH - IMS bridge header” on page 77.

ITSIC	'C'
ITSNIC	'I'
ITSARC	'A'

MD* (Message descriptor structure identifier)

See the *MDSID* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MDSIDV	'Mdbb'
--------	--------

MD* (Message descriptor version)

See the *MDVER* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MDVER1	1	X'00000001'
MDVER2	2	X'00000002'
MDVERC	(variable)	

ME* (Message descriptor extension length)

See the *MELEN* field described in “Chapter 11. MQMDE - Message descriptor extension” on page 131.

MELEN2	72	X'00000048'
--------	----	-------------

ME* (Message descriptor extension structure identifier)

See the *MESID* field described in “Chapter 11. MQMDE - Message descriptor extension” on page 131.

MESIDV	'MDEb'
--------	--------

ME* (Message descriptor extension version)

See the *MEVER* field described in “Chapter 11. MQMDE - Message descriptor extension” on page 131.

MEVER2	2	X'00000002'
MEVERC	2	X'00000002'

MEF* (Message descriptor extension flags)

See the *MEFLG* field described in “Chapter 11. MQMDE - Message descriptor extension” on page 131.

MEFNON	0	X'00000000'
--------	---	-------------

MS* (Message delivery sequence)

See the *MsgDeliverySequence* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

MSPRIO	0	X'00000000'
MSFIFO	1	X'00000001'

MF* (Message flags)

See the *MDMFL* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MFSEGI	0	X'00000000'
MFNONE	0	X'00000000'
MFSEGA	1	X'00000001'
MFSEG	2	X'00000002'
MFLSEG	4	X'00000004'
FMIG	8	X'00000008'
MFLMIG	16	X'00000010'

MF* (Message-flags masks)

See “Appendix E. Report options” on page 457.

MFAUM	-1048576	X'FFF00000'
MFAUXM	1044480	X'000FF000'
MFRUM	4095	X'0000FFFF'

MI* (Message identifier)

See the *MDMID* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MINONE	X'00...00' (24 nulls)
--------	-----------------------

MO* (Match options)

See the *GMMO* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

MONONE	0	X'00000000'
MOMSGI	1	X'00000001'
MOCORI	2	X'00000002'
MOGRPI	4	X'00000004'
MOSEQN	8	X'00000008'
MOOFFS	16	X'00000010'

MQSeries constants

MT* (Message type)

See the *MDMT* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MTSFST	1	X'00000001'
MTRQST	1	X'00000001'
MTRPLY	2	X'00000002'
MTRPRT	4	X'00000004'
MTDGRM	8	X'00000008'
MTSLST	65535	X'0000FFFF'
MTAFST	65536	X'00010000'
MTALST	99999999	X'3B9AC9FF'

MTK* (Message token)

See the *GMTOK* fields described in “Chapter 8. MQGMO - Get-message options” on page 51 and “Chapter 20. MQWIH - Work information header” on page 189.

MTKNON	X'00...00' (16 nulls)
--------	-----------------------

NC* (Name count)

See the *NameCount* attribute described in “Chapter 41. Attributes for namelists” on page 317.

NCMXNL	256	X'00000100'
--------	-----	-------------

OD* (Object descriptor length)

See “Chapter 12. MQOD - Object descriptor” on page 137.

ODLENC	336	X'00000150'
--------	-----	-------------

OD* (Object descriptor structure identifier)

See the *ODSID* field described in “Chapter 12. MQOD - Object descriptor” on page 137.

ODSIDV	'0Dbb'
--------	--------

OD* (Object descriptor version)

See the *ODVER* field described in “Chapter 12. MQOD - Object descriptor” on page 137.

ODVER1	1	X'00000001'
ODVER2	2	X'00000002'
ODVER3	3	X'00000003'
ODVERC	(variable)	

OII* (Object instance identifier)

See the *RMOII* field described in “Chapter 16. MQRMH - Message reference header” on page 169.

OIINON	X'00...00' (24 nulls)
--------	-----------------------

OL* (Original length)

See the *MDOLN* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

OLUNDF	-1	X'FFFFFFFF'
--------	----	-------------

OO* (Open options)

See the *OPTS* parameter described in “Chapter 32. MQOPEN - Open object” on page 251.

OOBNDQ	0	X'00000000'
OOINPQ	1	X'00000001'
OOINPS	2	X'00000002'
OOINPX	4	X'00000004'
OOBRW	8	X'00000008'
OOOUT	16	X'00000010'
OOINQ	32	X'00000020'
OOSSET	64	X'00000040'
OOSAVA	128	X'00000080'
OOPASI	256	X'00000100'
OOPASA	512	X'00000200'
OOSSETI	1024	X'00000400'
OOSSETA	2048	X'00000800'
OOALTU	4096	X'00001000'
OOFIQ	8192	X'00002000'
OOBNDO	16384	X'00004000'
OOBNDN	32768	X'00008000'

OT* (Object type)

See the *ODOT* field described in “Chapter 12. MQOD - Object descriptor” on page 137.

OTQ	1	X'00000001'
OTNLST	2	X'00000002'
OTPRO	3	X'00000003'
OTQM	5	X'00000005'

PE* (Persistence)

See the *MDPER* field described in “Chapter 10. MQMD - Message descriptor” on page 83, and the *DefPersistence* attribute described in “Chapter 37. Attributes for all queues” on page 293.

PENPER	0	X'00000000'
PEPER	1	X'00000001'

PL* (Platform)

See the *Platform* attribute described in “Chapter 43. Attributes for the queue manager” on page 323.

PLMVS	1	X'00000001'
PL390	1	X'00000001'
PLOS2	2	X'00000002'
PLAIX	3	X'00000003'

MQSeries constants

PLUNIX	3	X'00000003'
PL400	4	X'00000004'
PLWIN	5	X'00000005'
PLWINT	11	X'0000000B'
PLVMS	12	X'0000000C'
PLNSK	13	X'0000000D'

PM* (Put message options)

See the *PMOPT* field described in “Chapter 14. MQPMO - Put message options” on page 149.

PMNONE	0	X'00000000'
PMSYP	2	X'00000002'
PMNSYP	4	X'00000004'
PMDEFC	32	X'00000020'
PMNMID	64	X'00000040'
PMNCID	128	X'00000080'
PMPASI	256	X'00000100'
PMPASA	512	X'00000200'
PMSETI	1024	X'00000400'
PMSETA	2048	X'00000800'
PMALTU	4096	X'00001000'
PMFIQ	8192	X'00002000'
PMNOC	16384	X'00004000'
PMLOGO	32768	X'00008000'

PM* (Put message options structure length)

See “Chapter 14. MQPMO - Put message options” on page 149.

PMLENC	152	X'00000098'
--------	-----	-------------

PM* (Put message options structure identifier)

See the *PMSID* field described in “Chapter 14. MQPMO - Put message options” on page 149.

PMSIDV	'PMOb'
--------	--------

PM* (Put message options version)

See the *PMVER* field described in “Chapter 14. MQPMO - Put message options” on page 149.

PMVER1	1	X'00000001'
PMVER2	2	X'00000002'
PMVERC	(variable)	

PF* (Put message record field flags)

See the *DHPRF* field described in “Chapter 6. MQDH - Distribution header” on page 37.

PFNONE	0	X'00000000'
PFMID	1	X'00000001'
PFCID	2	X'00000002'

MQSeries constants

PFGID	4	X'00000004'
PFFB	8	X'00000008'
PFACC	16	X'00000010'

PR* (Priority)

See the *MDPRI* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

PRQDEF	-1	X'FFFFFFFF'
--------	----	-------------

QA* (Inhibit get)

See the *InhibitGet* attribute described in “Chapter 37. Attributes for all queues” on page 293.

QAGETA	0	X'00000000'
QAGETI	1	X'00000001'

QA* (Inhibit put)

See the *InhibitPut* attribute described in “Chapter 37. Attributes for all queues” on page 293.

QAPUTA	0	X'00000000'
QAPUTI	1	X'00000001'

QA* (Backout hardening)

See the *HardenGetBackout* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

QABNH	0	X'00000000'
QABH	1	X'00000001'

QA* (Queue shareability)

See the *Shareability* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

QANSHR	0	X'00000000'
QASHR	1	X'00000001'

QD* (Queue definition type)

See the *DefinitionType* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

QDPRE	1	X'00000001'
QDPERM	2	X'00000002'
QDTEMP	3	X'00000003'

QSIE* (Service interval events)

See the *QServiceIntervalEvent* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

MQSeries constants

QSIENO	0	X'00000000'
QSIEHI	1	X'00000001'
QSIEOK	2	X'00000002'

QT* (Queue type)

See the *QType* attribute described in “Chapter 37. Attributes for all queues” on page 293.

QTLOC	1	X'00000001'
QTMOD	2	X'00000002'
QTALS	3	X'00000003'
QTREM	6	X'00000006'
QTCLUS	7	X'00000007'

RC* (Reason code)

See “Appendix A. Return codes” on page 357, and the *MDFB* field described in “Chapter 10. MQMD - Message descriptor” on page 83. Note: the following list is in numeric order.

RCNONE	0	X'00000000'
RC0900	900	X'00000384'
RC0999	999	X'000003E7'
RC2001	2001	X'000007D1'
RC2002	2002	X'000007D2'
RC2003	2003	X'000007D3'
RC2004	2004	X'000007D4'
RC2005	2005	X'000007D5'
RC2006	2006	X'000007D6'
RC2007	2007	X'000007D7'
RC2008	2008	X'000007D8'
RC2009	2009	X'000007D9'
RC2010	2010	X'000007DA'
RC2011	2011	X'000007DB'
RC2013	2013	X'000007DD'
RC2014	2014	X'000007DE'
RC2016	2016	X'000007E0'
RC2017	2017	X'000007E1'
RC2018	2018	X'000007E2'
RC2019	2019	X'000007E3'
RC2020	2020	X'000007E4'
RC2021	2021	X'000007E5'
RC2022	2022	X'000007E6'
RC2023	2023	X'000007E7'
RC2024	2024	X'000007E8'
RC2026	2026	X'000007EA'
RC2027	2027	X'000007EB'
RC2029	2029	X'000007ED'
RC2030	2030	X'000007EE'
RC2031	2031	X'000007EF'
RC2033	2033	X'000007F1'
RC2034	2034	X'000007F2'
RC2035	2035	X'000007F3'
RC2036	2036	X'000007F4'
RC2037	2037	X'000007F5'
RC2038	2038	X'000007F6'
RC2039	2039	X'000007F7'

MQSeries constants

RC2040	2040	X'000007F8'
RC2041	2041	X'000007F9'
RC2042	2042	X'000007FA'
RC2043	2043	X'000007FB'
RC2044	2044	X'000007FC'
RC2045	2045	X'000007FD'
RC2046	2046	X'000007FE'
RC2047	2047	X'000007FF'
RC2048	2048	X'00000800'
RC2049	2049	X'00000801'
RC2050	2050	X'00000802'
RC2051	2051	X'00000803'
RC2052	2052	X'00000804'
RC2053	2053	X'00000805'
RC2055	2055	X'00000807'
RC2056	2056	X'00000808'
RC2057	2057	X'00000809'
RC2058	2058	X'0000080A'
RC2059	2059	X'0000080B'
RC2061	2061	X'0000080D'
RC2063	2063	X'0000080F'
RC2065	2065	X'00000811'
RC2066	2066	X'00000812'
RC2067	2067	X'00000813'
RC2068	2068	X'00000814'
RC2071	2071	X'00000817'
RC2072	2072	X'00000818'
RC2075	2075	X'0000081B'
RC2076	2076	X'0000081C'
RC2077	2077	X'0000081D'
RC2078	2078	X'0000081E'
RC2079	2079	X'0000081F'
RC2080	2080	X'00000820'
RC2082	2082	X'00000822'
RC2085	2085	X'00000825'
RC2086	2086	X'00000826'
RC2087	2087	X'00000827'
RC2090	2090	X'0000082A'
RC2091	2091	X'0000082B'
RC2092	2092	X'0000082C'
RC2093	2093	X'0000082D'
RC2094	2094	X'0000082E'
RC2095	2095	X'0000082F'
RC2096	2096	X'00000830'
RC2097	2097	X'00000831'
RC2098	2098	X'00000832'
RC2100	2100	X'00000834'
RC2101	2101	X'00000835'
RC2102	2102	X'00000836'
RC2104	2104	X'00000838'
RC2110	2110	X'0000083E'
RC2111	2111	X'0000083F'
RC2112	2112	X'00000840'
RC2113	2113	X'00000841'
RC2114	2114	X'00000842'
RC2115	2115	X'00000843'
RC2116	2116	X'00000844'
RC2117	2117	X'00000845'
RC2118	2118	X'00000846'

MQSeries constants

RC2119	2119	X'00000847'
RC2120	2120	X'00000848'
RC2123	2123	X'0000084B'
RC2124	2124	X'0000084C'
RC2125	2125	X'0000084D'
RC2126	2126	X'0000084E'
RC2135	2135	X'00000857'
RC2136	2136	X'00000858'
RC2137	2137	X'00000859'
RC2139	2139	X'0000085B'
RC2141	2141	X'0000085D'
RC2142	2142	X'0000085E'
RC2143	2143	X'0000085F'
RC2144	2144	X'00000860'
RC2145	2145	X'00000861'
RC2146	2146	X'00000862'
RC2148	2148	X'00000864'
RC2149	2149	X'00000865'
RC2150	2150	X'00000866'
RC2152	2152	X'00000868'
RC2153	2153	X'00000869'
RC2154	2154	X'0000086A'
RC2155	2155	X'0000086B'
RC2156	2156	X'0000086C'
RC2158	2158	X'0000086E'
RC2159	2159	X'0000086F'
RC2161	2161	X'00000871'
RC2162	2162	X'00000872'
RC2173	2173	X'0000087D'
RC2184	2184	X'00000888'
RC2185	2185	X'00000889'
RC2186	2186	X'0000088A'
RC2187	2187	X'0000088B'
RC2188	2188	X'0000088C'
RC2189	2189	X'0000088D'
RC2190	2190	X'0000088E'
RC2191	2191	X'0000088F'
RC2194	2194	X'00000892'
RC2195	2195	X'00000893'
RC2196	2196	X'00000894'
RC2197	2197	X'00000895'
RC2198	2198	X'00000896'
RC2199	2199	X'00000897'
RC2206	2206	X'0000089E'
RC2207	2207	X'0000089F'
RC2208	2208	X'000008A0'
RC2209	2209	X'000008A1'
RC2218	2218	X'000008AA'
RC2219	2219	X'000008AB'
RC2220	2220	X'000008AC'
RC2222	2222	X'000008AE'
RC2223	2223	X'000008AF'
RC2224	2224	X'000008B0'
RC2225	2225	X'000008B1'
RC2226	2226	X'000008B2'
RC2227	2227	X'000008B3'
RC2232	2232	X'000008B8'
RC2233	2233	X'000008B9'
RC2234	2234	X'000008BA'

MQSeries constants

RC2235	2235	X'000008BB'
RC2236	2236	X'000008BC'
RC2237	2237	X'000008BD'
RC2238	2238	X'000008BE'
RC2239	2239	X'000008BF'
RC2241	2241	X'000008C1'
RC2242	2242	X'000008C2'
RC2243	2243	X'000008C3'
RC2244	2244	X'000008C4'
RC2245	2245	X'000008C5'
RC2246	2246	X'000008C6'
RC2247	2247	X'000008C7'
RC2248	2248	X'000008C8'
RC2249	2249	X'000008C9'
RC2250	2250	X'000008CA'
RC2251	2251	X'000008CB'
RC2252	2252	X'000008CC'
RC2253	2253	X'000008CD'
RC2255	2255	X'000008CF'
RC2256	2256	X'000008D0'
RC2257	2257	X'000008D1'
RC2258	2258	X'000008D2'
RC2259	2259	X'000008D3'
RC2260	2260	X'000008D4'
RC2261	2261	X'000008D5'
RC2262	2262	X'000008D6'
RC2263	2263	X'000008D7'
RC2264	2264	X'000008D8'
RC2265	2265	X'000008D9'
RC2266	2266	X'000008DA'
RC2267	2267	X'000008DB'
RC2268	2268	X'000008DC'
RC2269	2269	X'000008DD'
RC2270	2270	X'000008DE'
RC2273	2273	X'000008E1'
RC2277	2277	X'000008E5'
RC2278	2278	X'000008E6'
RC2279	2279	X'000008E7'
RC2282	2282	X'000008EA'
RC2283	2283	X'000008EB'
RC2284	2284	X'000008EC'
RC2295	2295	X'000008F7'
RC2296	2296	X'000008F8'
RC2299	2299	X'000008FB'
RC2300	2300	X'000008FC'
RC2301	2301	X'000008FD'
RC2302	2302	X'000008FE'
RC2303	2303	X'000008FF'
RC2304	2304	X'00000900'
RC2305	2305	X'00000901'
RC2306	2306	X'00000902'
RC2307	2307	X'00000903'
RC2308	2308	X'00000904'
RC2309	2309	X'00000905'
RC2310	2310	X'00000906'
RC2311	2311	X'00000907'
RC2312	2312	X'00000908'
RC2313	2313	X'00000909'
RC2314	2314	X'0000090A'

MQSeries constants

RC2315	2315	X'0000090B'
RC2316	2316	X'0000090C'
RC2317	2317	X'0000090D'
RC2318	2318	X'0000090E'
RC2319	2319	X'0000090F'
RC2320	2320	X'00000910'
RC2321	2321	X'00000911'
RC2322	2322	X'00000912'
RC2323	2323	X'00000913'
RC2324	2324	X'00000914'
RC2325	2325	X'00000915'
RC2326	2326	X'00000916'
RC2327	2327	X'00000917'
RC2328	2328	X'00000918'
RC2329	2329	X'00000919'
RC2330	2330	X'0000091A'
RC2334	2334	X'0000091E'
RC2335	2335	X'0000091F'
RC2336	2336	X'00000920'
RC2337	2337	X'00000921'
RC2338	2338	X'00000922'
RC2339	2339	X'00000923'

RL* (Returned length)

See the *GMRL* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

RLUNDF	-1	X'FFFFFFFF'
--------	----	-------------

RM* (Reference message header structure identifier)

See the *RMSID* field described in “Chapter 16. MQRMH - Message reference header” on page 169.

RMSIDV	'RMHb'
--------	--------

RM* (Reference message header version)

See the *RMVER* field described in “Chapter 16. MQRMH - Message reference header” on page 169.

RMVER1	1	X'00000001'
RMVERC	1	X'00000001'

RM* (Reference message header flags)

See the *RMFLG* field described in “Chapter 16. MQRMH - Message reference header” on page 169.

RMNLST	0	X'00000000'
RMLAST	1	X'00000001'

RO* (Report options)

See the *MDREP* field described in “Chapter 10. MQMD - Message descriptor” on page 83.

MQSeries constants

RONMI	0	X'00000000'
ROCMTC	0	X'00000000'
RODLQ	0	X'00000000'
RONONE	0	X'00000000'
ROPAN	1	X'00000001'
RONAN	2	X'00000002'
ROPCI	64	X'00000040'
ROPMI	128	X'00000080'
ROCOA	256	X'00000100'
ROCOAD	768	X'00000300'
ROCOAF	1792	X'00000700'
ROCOD	2048	X'00000800'
ROCDD	6144	X'00001800'
ROCOF	14336	X'00003800'
ROEXP	2097152	X'00200000'
ROEXPD	6291456	X'00600000'
ROEXPF	14680064	X'00E00000'
ROEXC	16777216	X'01000000'
ROEXCD	50331648	X'03000000'
ROEXCF	117440512	X'07000000'
RODISC	134217728	X'08000000'

RO* (Report-options masks)

See “Appendix E. Report options” on page 457.

RORUM	270270464	X'101C0000'
ROAUM	-270532353	X'EFE000FF'
ROAUXM	261888	X'0003FF00'

SCO* (Queue scope)

See the *Scope* attribute described in “Chapter 37. Attributes for all queues” on page 293.

SCOQM	1	X'00000001'
SCOCEL	2	X'00000002'

SEG* (Segmentation)

See the *GMSEG* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

SEGIHB	'b'
SEGALW	'A'

SI* (Security identifier)

See the *ODASI* field described in “Chapter 12. MQOD - Object descriptor” on page 137.

SINONE	X'00...00' (40 nulls)
--------	-----------------------

SIT* (Security identifier type)

See the *ODASI* field described in “Chapter 12. MQOD - Object descriptor” on page 137.

MQSeries constants

SITNON	X'00'
SITWNT	X'01'

SP* (Syncpoint)

See the *SyncPoint* attribute described in “Chapter 43. Attributes for the queue manager” on page 323.

SPNAVL	0	X'00000000'
SPAVL	1	X'00000001'

SS* (Segment status)

See the *GMSST* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

SSNSEG	'b'
SSLSEG	'L'
SSSEG	'S'

TC* (Trigger control)

See the *TriggerControl* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

TCOFF	0	X'00000000'
TCON	1	X'00000001'

TM* (Trigger message structure identifier)

See the *TMSID* field described in “Chapter 18. MQTM - Trigger message” on page 179.

TMSIDV	'TMbb'
--------	--------

TM* (Trigger message version)

See the *TMVER* field described in “Chapter 18. MQTM - Trigger message” on page 179.

TMVER1	1	X'00000001'
TMVERC	1	X'00000001'

TC* (Trigger message character format structure identifier)

See the *TC2SID* field described in “Chapter 19. MQTMC2 - Trigger message (character format)” on page 185.

TCSIDV	'TMCb'
--------	--------

TC* (Trigger message character format version)

See the *TC2VER* field described in “Chapter 19. MQTMC2 - Trigger message (character format)” on page 185.

TT* (Trigger type)

See the *TriggerType* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

TTNONE	0	X'00000000'
TTFRST	1	X'00000001'
TTEVRY	2	X'00000002'
TTDPTH	3	X'00000003'

US* (Usage)

See the *Usage* attribute described in “Chapter 38. Attributes for local queues and model queues” on page 299.

USNORM	0	X'00000000'
USTRAN	1	X'00000001'

WI* (Wait interval)

See the *GMWI* field described in “Chapter 8. MQGMO - Get-message options” on page 51.

WIULIM	-1	X'FFFFFFFF'
--------	----	-------------

WI* (Workload information header flags)

See the *WIFLG* field described in “Chapter 20. MQWIH - Work information header” on page 189.

WINONE	0	X'00000000'
--------	---	-------------

WI* (Workload information header structure length)

See the *WILEN* field described in “Chapter 20. MQWIH - Work information header” on page 189.

WILEN1	120	X'00000078'
WILENC	120	X'00000078'

WI* (Workload information header structure identifier)

See the *WISID* field described in “Chapter 20. MQWIH - Work information header” on page 189.

WISIDV	'WIHb'
--------	--------

WI* (Workload information header version)

See the *WIVER* field described in “Chapter 20. MQWIH - Work information header” on page 189.

WIVER1	1	X'00000001'
WIVERC	1	X'00000001'

MQSeries constants

XR* (Data-conversion-exit response)

See the *DXRES* field described in “MQDXP – Data-conversion exit parameter” on page 468.

XROK	0	X'00000000'
XRFAIL	1	X'00000001'

XQ* (Transmission queue header structure identifier)

See the *XQSID* field described in “Chapter 21. MQXQH - Transmission queue header” on page 193.

XQSIDV	'XQHb'
--------	--------

XQ* (Transmission queue header version)

See the *XQVER* field described in “Chapter 21. MQXQH - Transmission queue header” on page 193.

XQVER1	1	X'00000001'
XQVERC	1	X'00000001'

Appendix C. Rules for validating MQI options

This appendix lists the situations that produce an RC2046 reason code from an MQOPEN, MQPUT, MQPUT1, MQGET, or MQCLOSE call.

MQOPEN call

For the options of the MQOPEN call:

- At least *one* of the following must be specified:

- OOBRW
- OOINPQ
- OOINPX
- OOINPS
- OOINQ
- OOOUT
- OOSET

- Only *one* of the following is allowed:

- OOINPQ
- OOINPX
- OOINPS

- Only *one* of the following is allowed:

- OOBNDQ
- OOBNDN
- OOBNDQ

Note: The options listed above are mutually exclusive. However, because the value of OOBNDQ is zero, specifying it with either of the other two bind options does not result in reason code RC2046. OOBNDQ is provided to aid program documentation.

- If OOSAVA is specified, one of the OOINP* options must also be specified.
- If one of the OOSET* or OOPAS* options is specified, OOOUT must also be specified.

MQPUT call

For the put-message options:

- The combination of PMSYP and PMNSYP is not allowed.

- Only *one* of the following is allowed:

- PMDEFC
- PMNOC
- PMPASA
- PMPASI
- PMSETA
- PMSETI

- PMALTU is not allowed (it is valid only on the MQPUT1 call).

MQI options

MQPUT1 call

For the put-message options, the rules are the same as for the MQPUT call, except for the following:

- PMALTU is allowed.
- PMLOGO is *not* allowed.

MQGET call

For the get-message options:

- Only *one* of the following is allowed:
 - GMNSYP
 - GMSYP
 - GMPSYP
- Only *one* of the following is allowed:
 - GMBRWF
 - GMBRWC
 - GMBRWN
 - GMMUC
- GMSYP is not allowed with any of the following:
 - GMBRWF
 - GMBRWC
 - GMBRWN
 - GMLK
 - GMUNLK
- GMPSYP is not allowed with any of the following:
 - GMBRWF
 - GMBRWC
 - GMBRWN
 - GMCMPM
 - GMUNLK
- If GMLK is specified, one of the following must also be specified:
 - GMBRWF
 - GMBRWC
 - GMBRWN
- If GMUNLK is specified, only the following are allowed:
 - GMNSYP
 - GMNWT

MQCLOSE call

For the options of the MQCLOSE call:

- The combination of CODEL and COPURG is not allowed.

Appendix D. Machine encodings

This appendix describes the structure of the *MDENC* field in the message descriptor (see “Chapter 10. MQMD - Message descriptor” on page 83).

The *MDENC* 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:

ENIMSK

Mask for binary-integer encoding.

This subfield occupies bit positions 28 through 31 within the *MDENC* field.

ENDMSK

Mask for packed-decimal-integer encoding.

This subfield occupies bit positions 24 through 27 within the *MDENC* field.

ENFMSK

Mask for floating-point encoding.

This subfield occupies bit positions 20 through 23 within the *MDENC* field.

ENRMSK

Mask for reserved bits.

This subfield occupies bit positions 0 through 19 within the *MDENC* field.

Binary-integer encoding

The following values are valid for the binary-integer encoding:

ENIUND

Undefined integer encoding.

Binary integers are represented using an encoding that is undefined.

ENINOR

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

ENIREV

Reversed integer encoding.

Machine encodings

Binary integers are represented in the same way as ENINOR, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as ENINOR.

Packed-decimal-integer encoding

The following values are valid for the packed-decimal-integer encoding:

ENDUND

Undefined packed-decimal encoding.

Packed-decimal integers are represented using an encoding that is undefined.

ENDNOR

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.

ENDREV

Reversed packed-decimal encoding.

Packed-decimal integers are represented in the same way as ENDNOR, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as ENDNOR.

Floating-point encoding

The following values are valid for the floating-point encoding:

ENFUND

Undefined floating-point encoding.

Floating-point numbers are represented using an encoding that is undefined.

ENFNOR

Normal IEEE float encoding.

Floating-point numbers are represented using the standard IEEE² floating-point format, with the bytes arranged as follows:

2. The Institute of Electrical and Electronics Engineers

- 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.

ENFREV

Reversed IEEE float encoding.

Floating-point numbers are represented in the same way as ENFNOR, but with the bytes arranged in reverse order. The bits within each byte are arranged in the same way as ENFNOR.

ENF390

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 *MDENC* field in MQMD, the relevant constants that describe the required encodings should be added together. Be sure to combine only one of the ENI* encodings with one of the END* encodings and one of the ENF* encodings.

Analyzing encodings

The *MDENC* field contains subfields; because of this, applications that need to examine the integer, packed decimal, or float encoding should use the technique described below.

Using arithmetic

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 *MDENC* 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 58.

Table 58. 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 E. Report options

This appendix concerns the *MDREP* and *MDMFL* fields that are part of the message descriptor MQMD specified on the MQGET, MQPUT, and MQPUT1 calls (see “Chapter 10. MQMD - Message descriptor” on page 83). 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 *MDREP* 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:

RORUM

Mask for unsupported report options that are rejected.

This mask identifies the bit positions within the *MDREP* 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 CCFAIL and reason code RC2061.

This subfield occupies bit positions 3, and 11 through 13.

ROAUM

Mask for unsupported report options that are accepted.

This mask identifies the bit positions within the *MDREP* 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 CCWARN with reason code RC2104 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:

ROCMTC
RODLQ
RODISC
ROEXC
ROEXCD
ROEXCF
ROEXP
ROEXPD

Report options

ROEXPF
RONAN
RONMI
RONONE
ROPAN
ROPCI
ROPMI

ROAUXM

Mask for unsupported report options that are accepted only in certain circumstances.

This mask identifies the bit positions within the *MDREP* 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 *ODMN* and *ODON* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code CCWARN with reason code RC2104 are returned if these conditions are satisfied, and CCFAIL with reason code RC2061 if not.

This subfield occupies bit positions 14 through 23.

The following report options are included in this subfield:

ROCOA
ROCOAD
ROCOAF
ROCOD
ROCODD
ROCODF

If there are any options specified in the *MDREP* 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 *MDREP* 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 CCWARN 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 *MDREP* field contains subfields; because of this, applications that need to check whether the sender of the message requested a particular report should use the technique described below.

Using arithmetic

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	ROCOA
COD	ROCOD
Exception	ROEXC
Expiration	ROEXP

Call the value A.

2. Divide the *MDREP* 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 ROEXC, test E for equality with each of the following to determine what was specified by the sender of the message:

RONONE
ROEXC
ROEXCD
ROEXCF

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 ROPMI or ROPCI 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 *MDMFL* 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 *MDMFL* 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

Report options

that bit 0 is the most significant bit, and bit 31 the least significant bit. The following masks are defined to identify the subfields:

MFRUM

Mask for unsupported message flags that are rejected.

This mask identifies the bit positions within the *MDMFL* 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 CCFAIL and reason code RC2249.

This subfield occupies bit positions 20 through 31.

The following message flags are included in this subfield:

- MFLMIG
- MFLSEG
- FMIG
- MFSEG
- MFSEGA

MFAUM

Mask for unsupported message flags that are accepted.

This mask identifies the bit positions within the *MDMFL* 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 CCOK.

This subfield occupies bit positions 0 through 11.

MFAUXM

Mask for unsupported message flags that are accepted only in certain circumstances.

This mask identifies the bit positions within the *MDMFL* 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 *ODMN* and *ODON* fields in the object descriptor specified on the MQOPEN or MQPUT1 call is not a local transmission queue).

Completion code CCOK is returned if these conditions are satisfied, and CCFAIL with reason code RC2249 if not.

This subfield occupies bit positions 12 through 19.

If there are flags specified in the *MDMFL* 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 *MDMFL* 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 F. 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 GMCONV option to be specified on the MQGET call.

The following are described:

- The processing performed by the queue manager in response to the GMCONV 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 462.
- Special considerations for the conversion of report messages; see “Conversion of report messages” on page 467.
- The parameters passed to the data-conversion exit; see “MQCONVX - Data conversion exit” on page 478.
- A call that can be used from the exit in order to convert character data between different representations; see “MQXCNVC - Convert characters” on page 473.
- The data-structure parameter which is specific to the exit; see “MQDXP - Data-conversion exit parameter” on page 468.

Conversion processing

The queue manager performs the following actions if the GMCONV 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 *MDCSI* and *MDENC* values in the control information in the message are identical to those in the *MSGDSC* 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 *MDCSI* and *MDENC* values in the *MSGDSC* 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
CCOK	RCNONE
CCWARN	RC2079
CCWARN	RC2080

Conversion processing

The following steps are performed only if the *MDCSI* or *MDENC* value in the control information in the message differs from that in the *MSGDSC* parameter, and there is data to be converted:

2. If the *MDFMT* field in the control information in the message has the value *FMNONE*, the message is returned unconverted, with completion code *CCWARN* and reason code *RC2110*.
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 *GMATM* option was *not* specified, the message is returned unconverted, with completion code *CCWARN* and reason code *RC2080*.
 - If the *GMATM* option was specified, the completion code is set to *CCWARN*, the reason code is set to *RC2079*, and conversion processing continues.
5. If the message can be accommodated in the buffer without truncation, or the *GMATM* 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 *CCWARN* and reason code *RC2110*.

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 by the *MQGET* call will usually be one of the following combinations:

Completion code	Reason code
CCOK	<i>RCNONE</i>
CCWARN	<i>RC2079</i>

However, if the conversion is performed by a user-written exit, other reason codes can be returned, even when the conversion is successful.

If the conversion fails (for whatever reason), the queue manager returns the unconverted message to the application, with the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter set to the values in the control information in the message, and with completion code *CCWARN*. 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:

FMADMN

FMCICS
 FMCMD1
 FMCMD2
 FMDLH
 FMDH
 FMEVNT
 FMIMS
 FMIMVS
 FMMDE
 FMPCF
 FMRMH
 FMSTR
 FMTM
 FMXQH

1. If the message expands during conversion, and exceeds the size of the *BUFFER* parameter, the following is done:
 - If the *GMATM* option was *not* specified, the message is returned unconverted, with completion code *CCWARN* and reason code *RC2120*.
 - If the *GMATM* option was specified, the message is truncated, the completion code is set to *CCWARN*, the reason code is set to *RC2079*, 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 *DATLEN* parameter is the length of the *unconverted* message before truncation.
5. When strings are converted between single-byte character sets (SBCS), double-byte character sets (DBCS), or multi-byte character sets (MBCS), the strings can expand or contract.
 - In the PCF formats *FMADMIN*, *FMEVNT*, and *FMPCF*, the strings in the *MQCFST* and *MQCFSL* structures expand or contract as necessary to accommodate the string after conversion.

For the string-list structure *MQCFSL*, the strings in the list may expand or contract by different amounts. If this happens, the queue manager pads the shorter strings with blanks to make them the same length as the longest string after conversion.
 - In the format *FMRMH*, the strings addressed by the *RMSEO*, *RMSNO*, *RMDEO*, and *RMDNO* fields expand or contract as necessary to accommodate the strings after conversion.
 - In the format *FMRFH*, the *RFNVS* field expands or contracts as necessary to accommodate the name/value pairs after conversion.

Processing conventions

- In structures with fixed field sizes, the queue manager allows strings to expand or contract within their fixed fields, provided that no significant information is lost. In this regard, trailing blanks and characters following the first null character in the field are treated as insignificant.
 - If the string expands, but only insignificant characters need to be discarded to accommodate the converted string in the field, the conversion succeeds and the call completes with CCOK and reason code RCNONE (assuming no other errors).
 - If the string expands, but the converted string requires significant characters to be discarded in order to fit in the field, the message is returned unconverted and the call completes with CCWARN and reason code RC2190.
 - If the string contracts, the queue manager pads the string with blanks to the length of the field.

6. For messages consisting of one or more MQ header structures followed by user data, it is possible for one or more of the header structures to be converted, while the remainder of the message is not. However, (with two exceptions) the *MDCSI* and *MDENC* fields in each header structure always correctly indicate the character set and encoding of the data that follows the header structure.

The two exceptions are the MQCIH and MQIIH structures, where the values in the *MDCSI* and *MDENC* fields in those structures are not significant. For those structures, the data following the structure is in the same character set and encoding as the MQCIH or MQIIH structure itself.

7. If the *MDCSI* or *MDENC* fields in the control information of the message being retrieved, or in the *MSGDSC* 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 *MDENC* 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 CCWARN and one of the RC2111, RC2112, RC2113, RC2114 or RC2115, RC2116, RC2117, RC2118 reason codes (as appropriate); the *MDCSI* and *MDENC* fields in the *MSGDSC* 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 *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are those specified by the application issuing the MQGET call.

8. In all cases, if the message is returned to the application unconverted the completion code is set to CCWARN, and the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter are set to the values appropriate to the unconverted data. This is done for FMNONE 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 *MDCSI* and *MDENC* fields in the *MSGDSC* parameter.)

When an error is diagnosed, either a specific reason code is returned, or the general reason code RC2119. The reason code returned depends on the diagnostic capabilities of the underlying data-conversion service.

9. If completion code CCWARN is returned, and more than one reason code is relevant, the order of precedence is as follows:
 - a. The following reason takes precedence over all others:
RC2079
 - b. Next in precedence is the following reason:
RC2110
 - c. The order of precedence within the remaining reason codes is not defined.
10. On completion of the MQGET call:
 - The following reason code indicates that the message was converted successfully:
RCNONE
 - The following reason code indicates that the message *may* have been converted successfully (check the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter to find out):
RC2079
 - 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:

11. With the exception of the following formats:
FMADMN
FMEVNT
FMIMVS
FMPCF
FMSTR

none of the built-in formats can be converted from or to character sets that do not have SBCS characters for the characters that are valid in queue names. If an attempt is made to perform such a conversion, the message is returned unconverted, with completion code CCWARN and reason code RC2111 or RC2115, as appropriate.

The Unicode character set UCS-2 is an example of a character set that does not have SBCS characters for the characters that are valid in queue names.

12. 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 FMADMN message, care must be taken to ensure that the application does not attempt to access data beyond the end of the data returned.
13. If the format name is FMDLH, 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 *DLFMT*, *DLCSI*, and *DLENC* 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 *DLCSI* and *DLENC* fields in the MQDLH structure

Processing conventions

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 *DLFMT* field in the MQDLH structure, or performs the conversion itself (if *DLFMT* is the name of a built-in format).

If the MQGET call returns a completion code of CCWARN, 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 *MDCSI* and *MDENC* fields in the *MSGDSC* parameter, and those in the MQDLH structure, in order to determine which of the above applies.

14. If the format name is FMXQH, 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 *MDFMT*, *MDCSI*, and *MDENC* fields in the MQMD structure contained *within* the MQXQH. For each subsequent MQ header structure present, the *MDFMT*, *MDCSI*, and *MDENC* 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 GMCONV option is specified for an FMXQH 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 *MDFMT*, *MDCSI*, and *MDENC* fields in the *MSGDSC* 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 GMCONV option specified must reset the *MDCSI* and *MDENC* fields in the *MSGDSC* parameter to the values desired for the application message data, prior to each MQGET call.

- The values of the *MDFMT*, *MDCSI*, and *MDENC* 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 *MSGDSC* 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 GMCONV 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 RO*D 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 RO*F, or specifies RO*D 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 *MDFMT* 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 GMCONV 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 RCNONE (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 CCOK and reason code RCNONE should be returned by the exit. The length of the message data to be converted is passed to the exit as the *INLEN* parameter.

MQDXP – Data-conversion exit parameter

The following table summarizes the fields in the structure.

Table 59. Fields in MQDXP

Field	Description	Page
<i>DXSID</i>	Structure identifier	468
<i>DXVER</i>	Structure version number	468
<i>DXAOP</i>	Application options	469
<i>DXENC</i>	Numeric encoding required by application	469
<i>DXCSI</i>	Character set required by application	469
<i>DXLEN</i>	Length in bytes of message data	469
<i>DXCC</i>	Completion code	470
<i>DXREA</i>	Reason code qualifying <i>DXCC</i>	470
<i>DXRES</i>	Response from exit	472
<i>DXHCN</i>	Connection handle	473

The MQDXP structure is a parameter that is passed to the data-conversion exit. See the description of the MQCONVX call for details of the data conversion exit.

Only the *DXLEN*, *DXCC*, *DXREA* and *DXRES* fields in MQDXP may be changed by the exit; changes to other fields are ignored. However, the *DXLEN* 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 XRFAIL in *DXRES*; however, the queue manager ignores the values of the *DXCC* and *DXREA* 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:

- *DXRES* field not XROK and not XRFAIL
- *DXCC* field not CCOK and not CCWARN
- *DXLEN* field less than zero, or *DXLEN* field changed when the message being converted is a segment that contains only part of a logical message.

Fields

DXSID (4-byte character string)
Structure identifier.

The value must be:

DXSIDV

Identifier for data conversion exit parameter structure.

This is an input field to the exit.

DXVER (10-digit signed integer)
Structure version number.

MQDXP - Data-conversion exit parameter

The value must be:

DXVER1

Version number for data-conversion exit parameter structure.

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

DXVERC

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 *DXVER* 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.

DXXOP (10-digit signed integer)

Reserved.

This is a reserved field; its value is 0.

DXAOP (10-digit signed integer)

Application options.

This is a copy of the *GMOPT* field of the MQGMO structure specified by the application issuing the MQGET call. The exit may need to examine these to ascertain whether the GMATM option was specified.

This is an input field to the exit.

DXENC (10-digit signed integer)

Numeric encoding required by application.

This is the numeric encoding required by the application issuing the MQGET call; see the *MDENC* field in the MQMD structure for more details.

If the conversion is successful, the exit should copy this to the *MDENC* field in the message descriptor.

This is an input field to the exit.

DXCSI (10-digit signed integer)

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 *MDCSI* field in the MQMD structure for more details. If the application specifies the special value CSQM 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 *MDCSI* field in the message descriptor.

This is an input field to the exit.

DXLEN (10-digit signed integer)

Length in bytes of message data.

MQDXP - Data-conversion exit parameter

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 *DXLEN*. The size of the message actually provided to the exit is always given by the *INLEN* parameter of the exit, irrespective of any truncation that may have occurred.

Truncation is indicated by the *DXREA* field having the value RC2079 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 *DATLEN* 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 *DXLEN* is always greater than zero.

This is an input/output field to the exit.

DXCC (10-digit signed integer)

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 CCWARN, 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 *CMPCOD* parameter of the MQGET call; only CCOK and CCWARN are valid. See the description of the *DXREA* field for recommendations on how the exit should set this field on output.

This is an input/output field to the exit.

DXREA (10-digit signed integer)

Reason code qualifying *DXCC*.

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 RC2079, indicating that the message was truncated

MQDXP - Data-conversion exit parameter

in order fit into the buffer provided by the application, and RC2119, 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 *DXREA* had the value RC2079 on input to the exit, the *DXREA* and *DXCC* fields should not be altered, irrespective of whether the conversion succeeds or fails.

(If the *DXCC* field is not CCOK, the application which retrieves the message can identify a conversion failure by comparing the returned *MDENC* and *MDCSI* 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, RC2079 should be returned in preference to any of the reasons that indicate conversion failure.)

- If *DXREA* had any other value on input to the exit:
 - If the conversion succeeds, *DXCC* should be set to CCOK and *DXREA* set to RCNONE.
 - If the conversion fails, or the message expands and has to be truncated to fit in the buffer, *DXCC* should be set to CCWARN (or left unchanged), and *DXREA* 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 GMATM option:

- If it did specify that option, reason RC2079 should be returned.
- If it did not specify that option, the message should be returned unconverted, with reason code RC2120.

The reason codes listed below are recommended for use by the exit to indicate the reason that conversion failed, but the exit can return other values from the set of RC* codes if deemed appropriate. In addition, the range of values RC0900 through RC0999 are allocated for use by the exit to indicate conditions that the exit wishes to communicate to the application issuing the MQGET call.

Note: If the message cannot be converted successfully, the exit *must* return XRFAIL in the *DXRES* field, in order to cause the queue manager to return the unconverted message. This is true regardless of the reason code returned in the *DXREA* field.

RC0900

(900, X'384') Lowest value for an application-defined reason code returned by a data-conversion exit.

RC0999

(999, X'3E7') Highest value for application-defined reason code returned by a data-conversion exit.

RC2120

(2120, X'848') Converted data too big for buffer.

RC2119

(2119, X'847') Message data not converted.

RC2111

(2111, X'83F') Source coded character set identifier not valid.

RC2113

(2113, X'841') Packed-decimal encoding in message not recognized.

MQDXP - Data-conversion exit parameter

RC2114

(2114, X'842') Floating-point encoding in message not recognized.

RC2112

(2112, X'840') Source integer encoding not recognized.

RC2115

(2115, X'843') Target coded character set identifier not valid.

RC2117

(2117, X'845') Packed-decimal encoding specified by receiver not recognized.

RC2118

(2118, X'846') Floating-point encoding specified by receiver not recognized.

RC2116

(2116, X'844') Target integer encoding not recognized.

RC2079

(2079, X'81F') Truncated message returned (processing completed).

This is an input/output field to the exit.

DXRES (10-digit signed integer)

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:

XROK 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 *DXCC* field on output from the exit
- The value of the *DXREA* field on output from the exit
- The value of the *DXLEN* field on output from the exit
- The contents of the exit's output buffer *OUTBUF*; the number of bytes returned is the lesser of the exit's *OUTLEN* parameter, and the value of the *DXLEN* field on output from the exit
- The value of the *MDENC* field in the exit's message descriptor parameter on output from the exit
- The value of the *MDCSI* field in the exit's message descriptor parameter on output from the exit

XRFAIL

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 *DXCC* field on output from the exit
- The value of the *DXREA* field on output from the exit
- The value of the *DXLEN* field on *input* to the exit
- The contents of the exit's input buffer *INBUF*; the number of bytes returned is given by the *INLEN* parameter

If the exit has altered *INBUF*, the results are undefined.

DXRES is an output field from the exit.

MQDXP - Data-conversion exit parameter

DXHCN (10-digit signed integer)
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.

RPG declaration (ILE)

```
D*..1.....2.....3.....4.....5.....6.....7..
D* MQDXP Structure
D*
D* Structure identifier
D DXSID                1         4
D* Structure version number
D DXVER                5         8I 0
D* Reserved
D DXXOP                9         12I 0
D* Application options
D DXAOP               13         16I 0
D* Numeric encoding required by application
D DXENC               17         20I 0
D* Character set required by application
D DXCSI               21         24I 0
D* Length in bytes of message data
D DXLEN               25         28I 0
D* Completion code
D DXCC                29         32I 0
D* Reason code qualifying CompCode
D DXREA               33         36I 0
D* Response from exit
D DXRES               37         40I 0
D* Connection handle
D DXHCN               41         44I 0
```

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.

Syntax

```
MQXCNVC (HCONN, OPTS, SRCCSI, SRCLLEN, SRCBUF, TGTCSI, TGTLEN,  
         TGTBUF, DATLEN, CMPCOD, REASON)
```

Parameters

HCONN (10-digit signed integer) – 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 *DXHCN* 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*:

MQXCNV — Convert characters

HCDEFH

Default connection handle.

OPTS (10-digit signed integer) – input
Options that control the action of MQXCNV.

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).

Default-conversion option: The following option controls the use of default character conversion:

DCCDEF

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 DCCDEF 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.

Padding option: The following option allows the queue manager to pad the converted string with blanks or discard insignificant trailing characters, in order to make the converted string fit the target buffer:

DCCFIL

Fill target buffer.

This option requests that conversion take place in such a way that the target buffer is filled completely:

- If the string contracts when it is converted, trailing blanks are added in order to fill the target buffer.
- If the string expands when it is converted, trailing characters that are not significant are discarded to make the converted string fit the target buffer. If this can be done successfully, the call completes with CCOK and reason code RCNONE.

If there are too few insignificant trailing characters, as much of the string as will fit is placed in the target buffer, and the call completes with CCWARN and reason code RC2120.

Insignificant characters are:

- Trailing blanks
- Characters following the first null character in the string (but excluding the first null character itself)

MQXCNV — Convert characters

- If the string, *TGTCSI*, and *TGTLEN* are such that the target buffer cannot be set completely with valid characters, the call fails with CCFAIL and reason code RC2144. This can occur when *TGTCSI* is a pure DBCS character set (such as UCS-2), but *TGTLEN* specifies a length that is an odd number of bytes.
- *TGTLEN* can be less than or greater than *SRLEN*. On return from MQXCNV, *DATLEN* has the same value as *TGTLEN*.

If this option is not specified:

- The string is allowed to contract or expand within the target buffer as required. Insignificant trailing characters are neither added nor discarded.

If the converted string fits in the target buffer, the call completes with CCOK and reason code RCNONE.

If the converted string is too big for the target buffer, as much of the string as will fit is placed in the target buffer, and the call completes with CCWARN and reason code RC2120. Note that fewer than *TGTLEN* bytes can be returned in this case.

- *TGTLEN* can be less than or greater than *SRLEN*. On return from MQXCNV, *DATLEN* is less than or equal to *TGTLEN*.

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, UCS-2 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.

Only one of the DCCS* values should be specified, combined with one of the DCCT* values:

DCCSNA

Source encoding is the default for the environment and programming language.

DCCSNO

Source encoding is normal.

DCCSRE

Source encoding is reversed.

DCCSUN

Source encoding is undefined.

DCCTNA

Target encoding is the default for the environment and programming language.

DCCTNO

Target encoding is normal.

DCCTRE

Target encoding is reversed.

MQXCNVN — Convert characters

DCCTUN

Target encoding is undefined.

The encoding values defined above can be added directly to the *OPTS* field. However, if the source or target encoding is obtained from the *MDENC* field in the MQMD or other structure, the following processing must be done:

1. The integer encoding must be extracted from the *MDENC* field by eliminating the float and packed-decimal encodings; see “Analyzing encodings” on page 455 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 *OPTS* field. These factors are:

DCCSFA

Factor for source encoding

DCCTFA

Factor for target encoding

If not specified, the encoding options default to undefined (DCC*UN). In most cases, this does not affect the successful completion of the MQXCNVN call. However, if the corresponding character set is a multibyte character set whose representation is dependent on the encoding (for example, a UCS-2 character set), the call fails with reason code RC2112 or RC2116 as appropriate.

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

DCCNON

No options specified.

DCCNON 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.

SRCCSI (10-digit signed integer) – input
Coded character set identifier of string before conversion.

This is the coded character set identifier of the input string in *SRCBUF*.

SRCLN (10-digit signed integer) – input
Length of string before conversion.

This is the length in bytes of the input string in *SRCBUF*; it must be zero or greater.

SRCBUF (1-byte character string×*SRCLN*) – input
String to be converted.

This is the buffer containing the string to be converted from one character set to another.

TGTCI (10-digit signed integer) – input
Coded character set identifier of string after conversion.

This is the coded character set identifier of the character set to which *SRCBUF* is to be converted.

TGTLEN (10-digit signed integer) – input
Length of output buffer.

MQXCNCV — Convert characters

This is the length in bytes of the output buffer *TGTBUF*; it must be zero or greater. It can be less than or greater than *SRCLLEN*.

TGTBUF (1-byte character string×*TGTLEN*) – output
String after conversion.

This is the string after it has been converted to the character set defined by *TGTCSI*. The converted string can be shorter or longer than the unconverted string. The *DATLEN* parameter indicates the number of valid bytes returned.

DATLEN (10-digit signed integer) – output
Length of output string.

This is the length of the string returned in the output buffer *TGTBUF*. The converted string can be shorter or longer than the unconverted string.

CMPCOD (10-digit signed integer) – output
Completion code.

It is one of the following:

CCOK
Successful completion.

CCWARN
Warning (partial completion).

CCFAIL
Call failed.

REASON (10-digit signed integer) – output
Reason code qualifying *CMPCOD*.

If *CMPCOD* is **CCOK**:

RCNONE
(0, X'000') No reason to report.

If *CMPCOD* is **CCWARN**:

RC2120
(2120, X'848') Converted data too big for buffer.

If *CMPCOD* is **CCFAIL**:

RC2010
(2010, X'7DA') Data length parameter not valid.

RC2150
(2150, X'866') DBCS string not valid.

RC2018
(2018, X'7E2') Connection handle not valid.

RC2046
(2046, X'7FE') Options not valid or not consistent.

RC2102
(2102, X'836') Insufficient system resources available.

RC2145
(2145, X'861') Source buffer parameter not valid.

RC2111
(2111, X'83F') Source coded character set identifier not valid.

RC2112
(2112, X'840') Source integer encoding not recognized.

RC2143
(2143, X'85F') Source length parameter not valid.

MQXCNV C — Convert characters

RC2071

(2071, X'817') Insufficient storage available.

RC2146

(2146, X'862') Target buffer parameter not valid.

RC2115

(2115, X'843') Target coded character set identifier not valid.

RC2116

(2116, X'844') Target integer encoding not recognized.

RC2144

(2144, X'860') Target length parameter not valid.

RC2195

(2195, X'893') Unexpected error occurred.

For more information on these reason codes, see "Appendix A. Return codes" on page 357.

RPG invocation

```
C*.1.....2.....3.....4.....5.....6.....7..
C          CALLP      MQXCNV C(HCONN : OPTS : SRCCSI :
C                               SRCLEN : SRCBUF : TGTCSI :
C                               TGTLEN : TGTBUF : DATLEN :
C                               CMPCOD : REASON)
```

The prototype definition for the call is:

```
D*.1.....2.....3.....4.....5.....6.....7..
DMQXCNV C          PR          EXTPROC('MQXCNV C')
D* Connection handle
D HCONN          10I 0 VALUE
D* Options that control the action of MQXCNV C
D OPTS          10I 0 VALUE
D* Coded character set identifier of string before conversion
D SRCCSI          10I 0 VALUE
D* Length of string before conversion
D SRCLEN          10I 0 VALUE
D* String to be converted
D SRCBUF          * VALUE
D* Coded character set identifier of string after conversion
D TGTCSI          10I 0 VALUE
D* Length of output buffer
D TGTLEN          10I 0 VALUE
D* String after conversion
D TGTBUF          * VALUE
D* Length of output string
D DATLEN          10I 0
D* Completion code
D CMPCOD          10I 0
D* Reason code qualifying CMPCOD
D REASON          10I 0
```

MQCONV X - Data conversion exit

This call definition describes the parameters that are passed to the data-conversion exit. No entry point called MQCONV X is actually provided by the queue manager (see usage note 11 on page 482).

This definition is part of the MQSeries Data Conversion Interface (DCI), which is one of the MQSeries framework interfaces.

Syntax

MQCONVX (*MQDXP*, *MQMD*, *INLEN*, *INBUF*, *OUTLEN*, *OUTBUF*)

Parameters

MQDXP (*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” on page 468 for details of the fields in this structure.

MQMD (*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 *MDFMT*, *MDENC*, and *MDCSI* of the unconverted message contained in *INBUF*.

Note: The *MQMD* 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 *MDVER* field in *MQMD* to verify that the fields that the exit needs to access are present in the structure.

On OS/400, the exit is passed a version-2 *MQMD*.

On output, the exit should change the *MDENC* and *MDCSI* 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.

INLEN (10-digit signed integer) – input
Length in bytes of *INBUF*.

This is the length of the input buffer *INBUF*, and specifies the number of bytes to be processed by the exit. *INLEN* 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.

INBUF (1-byte bit string×*INLEN*) – 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 *INBUF*; if this parameter is altered, the results are undefined.

OUTLEN (10-digit signed integer) – input
Length in bytes of *OUTBUF*.

MQCONVX — Data conversion exit

This is the length of the output buffer *OUTBUF*, 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.

OUTBUF (1-byte bit string×*OUTLEN*) – output
Buffer containing the converted message.

On output from the exit, if the conversion was successful (as indicated by the value *XROK* in the *DXRES* field of the *MQDXP* parameter), *OUTBUF* 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.

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 *GMCONV* option is specified on the MQGET call
 - The *MDFMT* field in the message descriptor is not *FMNONE*
 - The message is not already in the required representation; that is, one or both of the message's *MDCSI* and *MDENC* 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 *RCNONE* or *RC2079*
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 *GMATM* option on the MQGET call.
In this case, the *DXREA* field in the *MQDXP* parameter on input to the exit will have the value *RC2079*.
 - 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 467 for more details).
In this case, the *DXREA* field in the *MQDXP* parameter on input to the exit will have the value *RCNONE* (if the receiving application provided a buffer that was big enough for the message).

Thus the value of the *DXREA* 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 *INLEN* parameter will be *less than* the length implied by the format name contained in the *MDFMT* field in the message descriptor. The exit should therefore check the value of *INLEN* 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 *INLEN* is less than the value expected, the exit should return XRFAIL in the *DXRES* field of the *MQDXP* parameter, with the *DXCC* and *DXREA* fields set to CCWARN and RC2110 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 *INBUF*. If the conversion completes successfully, the exit should leave the *DXREA* field in the *MQDXP* parameter unchanged. This has the effect of returning RC2079 if the message was truncated by the receiver's queue manager, and RCNONE 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 *OUTBUF*. In this case the exit must decide whether to truncate the message; the *DXAOP* field in the *MQDXP* parameter will indicate whether the receiving application specified the GMATM option.

4. Generally it is recommended that all of the data in the message provided to the exit in *INBUF* 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 *OUTBUF* 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 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 *MDFMT* since the application 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 *MDCSI* or *MDENC* 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. *MDFMT* 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.

MQCONVX — Data conversion exit

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 RC2219, or other unpredictable errors.
11. No entry point called MQCONVX is actually provided by the queue manager. The name of the exit should be the same as the format name (the name contained in the *MDFMT* field in MQMD), although this is not required in all environments.

RPG invocation

```
C*..1.....2.....3.....4.....5.....6.....7..
C                               CALLP      exitname(MQDXP : MQMD : INLEN :
C                               INBUF : OUTLEN : OUTBUF)
```

The prototype definition for the call is:

```
D*..1.....2.....3.....4.....5.....6.....7..
Dexitname      PR              EXTPROC('exitname')
D* Data-conversion exit parameter block
D MQDXP                44A
D* Message descriptor
D MQMD                364A
D* Length in bytes of INBUF
D INLEN                10I 0 VALUE
D* Buffer containing the unconverted message
D INBUF                *  VALUE
D* Length in bytes of OUTBUF
D OUTLEN              10I 0 VALUE
D* Buffer containing the converted message
D OUTBUF                *  VALUE
```

End of product-sensitive programming interface

Appendix G. Notices

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

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

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

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

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

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

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

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

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

Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
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.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Programming interface information

This book is intended to help you write application programs that run under MQSeries for AS/400.

This book also documents General-use Programming Interface and Associated Guidance Information provided by MQSeries for AS/400 V5.1

General-use programming interfaces allow the customer to write programs that obtain the services of these products.

General-use Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	BookManager
CICS	CICS/VSE	FFST
IBM	IMS	MQ
MQSeries	OS/2	OS/390
OS/400	Presentation Manager	RACF
RPG/400	System/370	System/390

Lotus and LotusScript are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

Intel is a trademark of Intel Corporation in the United States and/or other countries.

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

Other company, product, or service names, may be the trademarks or service marks of others.

Applications

Glossary of terms and abbreviations

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

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

A

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

active log. See *recovery log*.

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

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

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

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

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

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

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

allied address space. See *ally*.

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

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

APAR. Authorized program analysis report.

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

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

application queue. A queue used by an application.

archive log. See *recovery log*.

ASID. Address space identifier.

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

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

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

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

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

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

B

backout. An operation that reverses all the changes made during the current unit of recovery or unit of

work. After the operation is complete, a new unit of recovery or unit of work begins. Contrast with *commit*.

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

BMS. Basic mapping support.

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

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

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

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

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

BSDS. Bootstrap data set.

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

C

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

CCF. Channel control function.

CCSID. Coded character set identifier.

CDF. Channel definition file.

channel. See *message channel*.

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

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

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

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

CI. Control interval.

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

CL. Control Language.

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

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

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

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

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

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

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

command processor. The MQSeries component that processes commands.

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

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

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

configuration file. In MQSeries on UNIX systems, MQSeries for 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 OS/390 before MQSeries for OS/390 has been started.

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

Distributed Computing Environment (DCE).

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

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

DLQ. Dead-letter queue.

DQM. Distributed queue management.

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

dual mode. See *dual logging*.

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

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

E

environment. See *application environment*.

ESM. External security manager.

ESTAE. Extended specify task abnormal exit.

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

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

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

event log. See *application log*.

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

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

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

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

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

F

FFST. First Failure Support Technology.

FIFO. First-in-first-out.

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

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

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

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

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

FRR. Functional recovery routine.

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

G

GCPC. Generalized command preprocessor.

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

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

get. In message queuing, to use the MQGET call to remove a message from a queue. See also *browse*.

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

GTF. Generalized Trace Facility.

H

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

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

ILE. Integrated Language Environment.

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

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

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

.ini file. See *configuration file*.

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

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

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

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

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

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

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

Interactive System Productivity Facility (ISPF). An IBM licensed program that serves as a full-screen editor

and dialog manager. It is used for writing application programs, and provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user.

IPCS. Interactive Problem Control System.

ISPF. Interactive System Productivity Facility.

L

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

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

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

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

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

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

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

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

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

log file. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a file in which all significant changes to the data controlled by a

queue manager are recorded. If the primary log files become full, MQSeries allocates secondary log files.

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

M

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

MCA. Message channel agent.

MCI. Message channel interface.

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

message. In message queuing applications, a communication sent between programs. See also *persistent message* and *nonpersistent message*. In system programming, information intended for the terminal operator or system administrator.

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

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

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

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

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

message queue. Synonym for *queue*.

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

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

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

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

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

MQAI. MQSeries Administration Interface.

MQI. Message queue interface.

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

MQSC. MQSeries commands.

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

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

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

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects. Contrast with *programmable command format (PCF)*.

N

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

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

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

name transformation. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows

NT, an internal process that changes a queue manager name so that it is unique and valid for the system being used. Externally, the queue manager name remains unchanged.

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

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

NSI. Name service interface.

NTFS. New Technology File System.

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

O

OAM. Object authority manager.

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

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

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

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

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

OPM. Original Program Model.

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

OTMA. Open Transaction Manager Access.

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

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

P

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

PCF. Programmable command format.

PCF command. See *programmable command format*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Contrast with *MQSC*.

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

PTF. Program temporary fix.

Q

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

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

queue manager event. An event that indicates:

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

queuing. See *message queuing*.

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

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

R

RBA. Relative byte address.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Resource Recovery Services (RRS). An OS/390 facility that provides 2-phase syncpoint support across participating resource managers.

responder. In distributed queuing, a program that replies to network connection requests from another system.

resynch. In MQSeries, an option to direct a channel to start up and resolve any in-doubt status messages, but without restarting message transfer.

return codes. The collective name for completion codes and reason codes.

rollback. Synonym for *back out*.

RRS. Resource Recovery Services.

RTM. Recovery termination manager.

rules table. A control file containing one or more rules that the dead-letter queue handler applies to messages on the DLQ.

S

SAF. System Authorization Facility.

SDWA. System diagnostic work area.

security enabling interface (SEI). The MQSeries interface to which customer- or vendor-written programs that check authorization, supply a user identifier, or perform authentication must conform. A part of the MQSeries Framework.

SEI. Security enabling interface.

sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

sequential delivery. In MQSeries, a method of transmitting messages with a sequence number so that the receiving channel can reestablish the message sequence when storing the messages. This is required where messages must be delivered only once, and in the correct order.

sequential number wrap value. In MQSeries, a method of ensuring that both ends of a communication link reset their current message sequence numbers at the same time. Transmitting messages with a sequence number ensures that the receiving channel can reestablish the message sequence when storing the messages.

server. (1) In MQSeries, a queue manager that provides queue services to client applications running on a remote workstation. (2) The program that responds to requests for information in the particular two-program, information-flow model of client/server. See also *client*.

server channel. In message queuing, a channel that responds to a requester channel, removes messages from a transmission queue, and moves them over a communication link to the requester channel.

server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *client connection channel type*.

service interval. A time interval, against which the elapsed time between a put or a get and a subsequent get is compared by the queue manager in deciding whether the conditions for a service interval event have been met. The service interval for a queue is specified by a queue attribute.

service interval event. An event related to the service interval.

session ID. In MQSeries for OS/390, the CICS-unique identifier that defines the communication link to be

used by a message channel agent when moving messages from a transmission queue to a link.

shutdown. See *immediate shutdown*, *preemptive shutdown*, and *quiesced shutdown*.

signaling. In MQSeries for OS/390 and MQSeries for Windows 2.1, a feature that allows the operating system to notify a program when an expected message arrives on a queue.

single logging. A method of recording MQSeries for OS/390 activity where each change is recorded on one data set only. Contrast with *dual logging*.

single-phase backout. A method in which an action in progress must not be allowed to finish, and all changes that are part of that action must be undone.

single-phase commit. A method in which a program can commit updates to a queue without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with *two-phase commit*.

SIT. System initialization table.

stanza. A group of lines in a configuration file that assigns a value to a parameter modifying the behavior of a queue manager, client, or channel. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a configuration (.ini) file may contain a number of stanzas.

storage class. In MQSeries for OS/390, a storage class defines the page set that is to hold the messages for a particular queue. The storage class is specified when the queue is defined.

store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

subsystem. In OS/390, a group of modules that provides function that is dependent on OS/390. For example, MQSeries for OS/390 is an OS/390 subsystem.

supervisor call (SVC). An OS/390 instruction that interrupts a running program and passes control to the supervisor so that it can perform the specific service indicated by the instruction.

SVC. Supervisor call.

switch profile. In MQSeries for OS/390, a RACF profile used when MQSeries starts up or when a refresh security command is issued. Each switch profile that MQSeries detects turns off checking for the specified resource.

symptom string. Diagnostic information displayed in a structured format designed for searching the IBM software support database.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

syncpoint. An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

System Authorization Facility (SAF). An OS/390 facility through which MQSeries for OS/390 communicates with an external security manager such as RACF.

system.command.input queue. A local queue on which application programs can put MQSeries commands. The commands are retrieved from the queue by the command server, which validates them and passes them to the command processor to be run.

system control commands. Commands used to manipulate platform-specific entities such as buffer pools, storage classes, and page sets.

system diagnostic work area (SDWA). Data recorded in a SYS1.LOGREC entry, which describes a program or hardware error.

system initialization table (SIT). A table containing parameters used by CICS on start up.

SYS1.LOGREC. A service aid containing information about program and hardware errors.

T

TACL. Tandem Advanced Command Language.

target library high-level qualifier (thlqual). High-level qualifier for OS/390 target data set names.

task control block (TCB). An OS/390 control block used to communicate information about tasks within an address space that are connected to an OS/390 subsystem such as MQSeries for OS/390 or CICS.

task switching. The overlapping of I/O operations and processing between several tasks. In MQSeries for OS/390, the task switcher optimizes performance by allowing some MQI calls to be executed under subtasks rather than under the main CICS TCB.

TCB. Task control block.

temporary dynamic queue. A dynamic queue that is deleted when it is closed. Temporary dynamic queues are not recovered if the queue manager fails, so they can contain nonpersistent messages only. Contrast with *permanent dynamic queue*.

termination notification. A pending event that is activated when a CICS subsystem successfully connects to MQSeries for OS/390.

thlqual. Target library high-level qualifier.

thread. In MQSeries, the lowest level of parallel execution available on an operating system platform.

time-independent messaging. See *asynchronous messaging*.

TMI. Trigger monitor interface.

trace. In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF). See also *global trace* and *performance trace*.

tranid. See *transaction identifier*.

transaction identifier. In CICS, a name that is specified when the transaction is defined, and that is used to invoke the transaction.

transmission program. See *message channel agent*.

transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

trigger event. An event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

triggering. In MQSeries, a facility allowing a queue manager to start an application automatically when predetermined conditions on a queue are satisfied.

trigger message. A message containing information about the program that a trigger monitor is to start.

trigger monitor. A continuously-running application serving one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. It uses the information in the trigger message to start a process that serves the queue on which a trigger event occurred.

trigger monitor interface (TMI). The MQSeries interface to which customer- or vendor-written trigger monitor programs must conform. A part of the MQSeries Framework.

two-phase commit. A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with *single-phase commit*.

U

UIS. User identifier service.

undelivered-message queue. See *dead-letter queue*.

undo/redo record. A log record used in recovery. The redo part of the record describes a change to be made to an MQSeries object. The undo part describes how to back out the change if the work is not committed.

unit of recovery. A recoverable sequence of operations within a single resource manager. Contrast with *unit of work*.

unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction. Contrast with *unit of recovery*.

user identifier service (UIS). In MQSeries for OS/2 Warp, the facility that allows MQI applications to associate a user ID, other than the default user ID, with MQSeries messages.

utility. In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

Bibliography

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

MQSeries cross-platform publications

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

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS V2.2.1.1
- MQSeries for DIGITAL UNIX (Compaq Tru64 UNIX) V2.2.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for Sun Solaris V5.1
- MQSeries for Tandem NonStop Kernel V2.2.0.1
- MQSeries for VSE/ESA V2.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.1

Any exceptions to this general rule are indicated.

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

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 Queue Manager Clusters

MQSeries Queue Manager Clusters, SC34-5349, describes MQSeries clustering. It explains the concepts and terminology and shows how you can benefit by taking advantage of clustering. It details changes to the MQI, and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

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, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:

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

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 Format (PCF) messages, and installable services.

MQSeries Administration Interface Programming Guide and Reference

The *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390, provides information for users of the MQAI. The MQAI is a programming interface that simplifies the way in which applications manipulate Programmable Command Format (PCF) messages and their associated data structures.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Messages

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

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

- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

For other MQSeries platforms, the messages are supplied with the system. They do not appear in softcopy manual form.

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 these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for AS/400 V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries C++ is also supported by MQSeries clients supplied with these products and installed in the following environments:

- AIX
- HP-UX

- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 and Windows 98

MQSeries Using Java

MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Bindings for Java and the MQSeries Client for Java. MQSeries classes for Java are supported by these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for MVS/ESA V1.2
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

This book is available in softcopy only.

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 V5.1 Quick Beginnings, GC33-1867

MQSeries for AS/400

MQSeries for AS/400 V5.1 Quick Beginnings, GC34-5557

MQSeries for AS/400 V5.1 System Administration, SC34-5558

MQSeries for AS/400 V5.1 Application Programming Reference (ILE RPG), SC34-5559

MQSeries for AT&T GIS UNIX

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

MQSeries for Compaq (DIGITAL) OpenVMS

MQSeries for Digital OpenVMS System Management Guide, GC33-1791

MQSeries for Digital UNIX (Compaq Tru64 UNIX)

MQSeries for Digital UNIX System Management Guide, GC34-5483

MQSeries for HP-UX

MQSeries for HP-UX V5.1 Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp V5.1 Quick Beginnings, GC33-1868

MQSeries for OS/390

MQSeries for OS/390 Version 2 Release 1 Licensed Program Specifications, GC34-5377

MQSeries for OS/390 Version 2 Release 1 Program Directory

MQSeries for OS/390 System Management Guide, SC34-5374

MQSeries for OS/390 Messages and Codes, GC34-5375

MQSeries for OS/390 Problem Determination Guide, GC34-5376

MQSeries link for R/3

MQSeries link for R/3 Version 1.2 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

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

MQSeries for Sun Solaris

MQSeries for Sun Solaris V5.1 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel

MQSeries for Tandem NonStop Kernel System Management Guide, GC33-1893

MQSeries for VSE/ESA

MQSeries for VSE/ESA Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA System Management Guide, GC34-5364

MQSeries for Windows

MQSeries for Windows V2.0 User's Guide, GC33-1822

MQSeries for Windows V2.1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT V5.1 Quick Beginnings, GC34-5389

MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387

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

HTML format

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

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1 (compiled HTML)
- MQSeries link for R/3 V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

<http://www.ibm.com/software/ts/mqseries/>

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

<http://www.adobe.com/>

PDF versions of relevant MQSeries books are supplied with these MQSeries products:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1

- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries link for R/3 V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

<http://www.ibm.com/software/ts/mqseries/>

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

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

MQSeries information available on the Internet

The MQSeries product family Web site is at:

<http://www.ibm.com/software/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 and PDF formats.
- Download MQSeries SupportPacs.

Related publications

- AS/400 ILE RPG/400 Programmer's Guide*, SC09-1525
- AS/400 ILE RPG/400 Reference*, SC09-1526

Index

A

- AC* values 116
- alias queue 315
- aliasing
 - queue manager 313
 - reply queue 313
- AlterationDate attribute
 - namelist 317
 - process definition 319
 - queue 293
 - queue manager 324
- AlterationTime attribute
 - namelist 317
 - process definition 319
 - queue 293
 - queue manager 324
- AMQ3ECH4 sample program 348
- AMQ3GBR4 sample program 344
- AMQ3GET4 sample program 345
- AMQ3INQ4 sample program 349
- AMQ3PUT4 sample program 343
- AMQ3REQ4 sample program 346
- AMQ3SET4 sample program 351
- AMQ3SRV4 sample program 352
- AMQ3TRG4 sample program 352
- AppId attribute 319
- AppType attribute 320
- AT* values
 - AppType attribute 320
 - MDPAT field 117
 - TMAT field 181
- attributes
 - alias queue 315
 - common to all queues 293
 - local queue 299
 - namelist 317
 - process definition 319
 - queue manager 323
 - remote queue, local definition of 313
- AuthorityEvent attribute 324

B

- BackoutRequeueQName attribute 300
- BackoutThreshold attribute 300
- BaseQName attribute 315
- begin options structure 15
- BEGOP parameter 207
- bibliography 499
- BND* values 294
- BO* values 15
- BookManager 502
- BOOPT field 15
- BOSID field 15
- BOVER field 15
- BUFFER parameter
 - MQGET call 232
 - MQPUT call 266
 - MQPUT1 call 276
- BUFLEN parameter
 - MQGET call 232

- BUFLEN parameter (*continued*)
 - MQPUT call 265
 - MQPUT1 call 276
- building your application 337
- built-in formats 104

C

- CA* values 242
- CALEN parameter
 - MQINQ call 246
 - MQSET call 285
- calls
 - conventions used 201
 - detailed description
 - MQBACK 203
 - MQBEGIN 207
 - MQCLOSE 211
 - MQCMIT 217
 - MQCONN 221
 - MQCONNX 227
 - MQCONVX 478
 - MQDISC 229
 - MQGET 231
 - MQINQ 241
 - MQOPEN 251
 - MQPUT 265
 - MQPUT1 275
 - MQSET 283
 - MQXCNCV 473
 - CC* values 357
 - CF* values 24
 - ChannelAutoDef attribute 324
 - ChannelAutoDefEvent attribute 325
 - ChannelAutoDefExit attribute 325
 - CHRATR parameter
 - MQINQ call 247
 - MQSET call 285
 - CI* values 19, 112
 - CIAC field 25
 - CIADS field 23
 - CIAI field 26
 - CIAUT field 25
 - CICC field
 - MQCIH structure 21
 - CICNC field 26
 - CICP field 27
 - CICSI field 19
 - CICT field 23
 - CIENC field 19
 - CIEO field 27
 - CIFAC field 24
 - CIFKT field 22
 - CIFL field 26
 - CIFLG field 20
 - CIFMT field 20
 - CIFNC field 24
 - CIGWI field 21
 - CIII field 27
 - CILEN field 19
 - CILT field 22
 - CINTI field 27
- CIODL field 22
- CIREA field 21
- CIRET field 20
- CIRFM field 25
- CIRS1 field 25
- CIRS2 field 27
- CIRS3 field 27
- CIRS4 field 27
- CIRSI field 25
- CIRTI field 25
- CISC field 26
- CISID field 19
- CITES field 23
- CITI field 25
- CIUOW field 21
- CIVER field 19
- ClusterName attribute 293
- ClusterNamelist attribute 294
- ClusterWorkloadData attribute 325
- ClusterWorkloadExit attribute 325
- ClusterWorkloadLength attribute 326
- CMLV* values 327
- CMPCOD parameter
 - MQBEGIN call 207
 - MQCLOSE call 213
 - MQCONN call 223
 - MQCONNX call 227
 - MQDISC call 229
 - MQGET call 233
 - MQINQ call 247
 - MQOPEN call 257
 - MQPUT call 267
 - MQPUT1 call 276
 - MQSET call 285
 - MQXCNCV call 477
- CN* values 31, 32
- CNCCO field 33
- CNCCP field 33
- CNOPT field 32
- CNOPT parameter 227
- CNSID field 31
- CNVER field 31
- CO* values 211
- coded character set identifier 326
- CodedCharSetId attribute 326
- COMCOD parameter
 - MQBACK call 203
 - MQCMIT call 217
- CommandInputQName attribute 326
- CommandLevel attribute 327
- compatibility mode 224
- compiling 337
- completion code 357
- connect options structure 31
- constants, values of 421
 - accounting token (AC*) 422
 - accounting token type (ATT*) 422
 - application type (AT*) 423
 - backout hardening (QA*) 441
 - begin options (BO*) 423

constants, values of 421 (*continued*)

- begin options structure identifier (BO*) 423
- begin options version (BO*) 423
- binding (BND*) 423
- call identifier (MQ*) 426
- character attribute selectors (CA*) 424
- CICS bridge return code (CRC*) 427
- CICS function name (CF*) 425
- CICS header ADS descriptor (AD*) 424
- CICS header conversational task (CT*) 425
- CICS header facility (FC*) 425
- CICS header flags (CIF*) 426
- CICS header get-wait interval (WI*) 425
- CICS header length (CI*) 426
- CICS header link type (LT*) 426
- CICS header output data length (OL*) 427
- CICS header structure identifier (CI*) 426
- CICS header task end status (TE*) 428
- CICS header transaction start code (SC*) 428
- CICS header unit-of-work control (CU*) 428
- CICS header version (CI*) 426
- close options (CO*) 427
- coded character set identifier (CS*) 425
- command level (CMLV*) 427
- completion codes (CC*) 425
- connect options (CN*) 427
- connect options structure identifier (CN*) 427
- connect options version (CN*) 427
- connection handle (HC*) 433
- convert-characters masks and factors (DCC*) 428
- convert-characters options (DCC*) 428
- correlation identifier (CI*) 425
- data-conversion-exit parameter structure identifier (DX*) 430
- data-conversion-exit parameter structure version (DX*) 430
- data-conversion-exit response (XR*) 450
- dead-letter header structure identifier (DL*) 429
- dead-letter header version (DL*) 429
- distribution header flags (DHF*) 429
- distribution header structure identifier (DH*) 429
- distribution header version (DH*) 429
- distribution list support (DL*) 429
- encoding (EN*) 430
- encoding for binary integers (EN*) 431
- encoding for floating-point numbers (EN*) 430

constants, values of 421 (*continued*)

- encoding for packed-decimal integers (EN*) 430
- encoding masks (EN*) 430
- event reporting (EV*) 431
- event reporting (QSIE*) 441
- expiry interval (EI*) 430
- feedback (FB*) 431
- format (FM*) 432
- get message options (GM*) 432
- get message options structure identifier (GM*) 433
- get message options version (GM*) 433
- group identifier (GI*) 432
- group status (GS*) 433
- IMS authenticator (IAU*) 435
- IMS commit mode (ICM*) 435
- IMS header flags (II*) 435
- IMS header length (II*) 435
- IMS header structure identifier (II*) 435
- IMS header version (II*) 435
- IMS security scope (ISS*) 435
- IMS transaction instance identifier (ITI*) 436
- IMS transaction state (ITS*) 436
- inhibit get (QA*) 441
- inhibit put (QA*) 441
- integer attribute selectors (IA*) 433
- integer attribute value (IAV*) 435
- lengths of character string and byte fields (LN*) 421
- match options (MO*) 437
- message delivery sequence (MS*) 437
- message descriptor extension flags (MEF*) 437
- message descriptor extension length (ME*) 436
- message descriptor extension structure identifier (ME*) 436
- message descriptor extension version (ME*) 436
- message descriptor structure identifier (MD*) 436
- message descriptor version (MD*) 436
- message flags (MF*) 437
- message-flags masks (MF*) 437
- message identifier (MI*) 437
- message token (MTK*) 438
- message type (MT*) 438
- name count (NC*) 438
- object descriptor length (OD*) 438
- object descriptor structure identifier (OD*) 438
- object descriptor version (OD*) 438
- object handle (HO*) 433
- object instance identifier (OII*) 438
- object type (OT*) 439
- open options (OO*) 439
- original length (OL*) 439
- persistence (PE*) 439
- platform (PL*) 439
- priority (PR*) 441
- put message options (PM*) 440

constants, values of 421 (*continued*)

- put message options length (PM*) 440
- put message options structure identifier (PM*) 440
- put message options version (PM*) 440
- put message record field flags (PF*) 440
- queue definition type (QD*) 441
- queue shareability (QA*) 441
- queue type (QT*) 442
- reason codes (RC*) 442
- reference message header flags (RM*) 446
- reference message header structure identifier (RM*) 446
- reference message header version (RM*) 446
- report options (RO*) 446
- report-options masks (RO*) 447
- returned length (RL*) 446
- scope (SCO*) 447
- security identifier (SI*) 447
- security identifier type (SIT*) 447
- segment status (SS*) 448
- segmentation (SEG*) 447
- syncpoint (SP*) 448
- transmission queue header structure identifier 450
- transmission queue header version (XQ*) 450
- trigger controls (TC*) 448
- trigger message (character format) structure identifier (TC*) 448
- trigger message (character format) version (TC*) 448
- trigger message structure identifier (TM*) 448
- trigger message version (TM*) 448
- trigger type (TT*) 449
- undelivered-message header structure identifier (DL*) 429
- undelivered-message header version (DL*) 429
- usage (US*) 449
- wait interval (WI*) 449
- workload information header flags (WI*) 449
- workload information header structure identifier (WI*) 449
- workload information header structure length (WI*) 449
- workload information header version (WI*) 449
- conversion of report messages 467
- copy file – RPG programming language 8
- copy files 337
- CRC* values 20
- CreationDate attribute 300
- CreationTime attribute 300
- CRTPGM 337
- CRTRPGMOD 337
- CRTRPGPGM 337
- CS* values 103
- CU* values 21

CurrentQDepth attribute 301

D

data conversion

processing conventions 462
report messages 467

data types, conventions used 3, 7

data types, detailed description

elementary

ILE 5
MQBYTE 3
MQBYTE_n 4
MQCHAR 4
MQCHAR_n 4
MQHCONN 4
MQHOBJ 5
MQLONG 5
overview 3

structure

MQBO 15
MQCIH 17
MQCNO 31
MQDH 37
MQDLH 43
MQDXP 468
MQGMO 51
MQIIH 77
MQMD 83
MQMDE 131
MQOD 137
MQOR 147
MQPMO 149
MQPMR 165
MQRMH 169
MQRR 177
MQTM 179
MQTMC2 185
MQWIH 189
MQXQH 193
overview of 7

DATLEN parameter

MQGET call 232
MQXCNCV call 477

DCC* values 474

dead-letter header structure 43

DeadLetterQName attribute 327

DefBind attribute 294

DefinitionType attribute 301

DefInputOpenOption attribute 302

DefPersistence attribute 294

DefPriority attribute 295

DefXmitQName attribute 328

DH* values 38

DHCNT field 40

DHCSI field 39

DHENC field 39

DHF* values 39

DHFLG field 39

DHFMT field 39

DHLEN field 38

DHORO field 40

DHPRF field 40

DHPRO field 40

DHSID field 38

DHVER field 38

DistLists attribute 302, 328

distribution header structure 37

distribution lists 302, 328

DL* values 45, 302, 328

DLCSI field 47

DLDM field 46

DLDQ field 46

DLENC field 46

DLFMT field 47

DLPAN field 48

DLPAT field 47

DLPD field 48

DLPT field 48

DLREA field 45

DLSID field 45

DLVER field 45

DX* values 468

DXAOP field 469

DXCC field 470

DXCSI field 469

DXENC field 469

DXHCN field 473

DXLEN field 470

DXREA field 470

DXRES field 472

DXSID field 468

DXVER field 469

DXXOP field 469

dynamic queue 251

E

EI* values 98

EN* values 102

Encoding field

using 453

EnvData attribute 320

EV* values

AuthorityEvent attribute 324

ChannelAutoDefEvent attribute 325

InhibitEvent attribute 329

LocalEvent attribute 329

PerformanceEvent attribute 331

QDepthHighEvent attribute 307

QDepthLowEvent attribute 307

QDepthMaxEvent attribute 308

RemoteEvent attribute 332

StartStopEvent attribute 333

F

FB* values 45, 98

FM* values 104

fonts in this book xii

formats built-in 104

G

get-message options structure 51

GI* values 122

glossary 487

GM* values 51, 52

GMGST field 72

GMMO field 70

GMO parameter 232

GMOPT field 52

GMRE1 field 73

GMRL field 74

GMRQN field 70

GMSEG field 73

GMSG1 field 70

GMSG2 field 70

GMSID field 51

GMSST field 73

GMTOK field 73

GMVER field 52

GMWI field 69

GS* values 72

H

handle scope 223, 257

handles 329

HardenGetBackout attribute 303

HC* values 229

HCONN parameter

MQBACK call 203

MQBEGIN call 207

MQCLOSE call 211

MQCMIT call 217

MQCONN call 222

MQCONN_X call 227

MQDISC call 229

MQGET call 231

MQINQ call 241

MQOPEN call 251

MQPUT call 265

MQPUT1 call 275

MQSET call 283

MQXCNCV call 473

scope 223

HO* values 211

HOBJ parameter

MQCLOSE call 211

MQGET call 231

MQINQ call 241

MQOPEN call 257

MQPUT call 265

MQSET call 283

scope 257

HTML (Hypertext Markup

Language) 502

Hypertext Markup Language

(HTML) 502

I

IA* values 242, 284

IACNT parameter

MQINQ call 246

MQSET call 284

IAU* values 79

IAV* values 246

ICM* values 80

II* values 78

IIAUT field 79

IICMT field 80

IICSI field 78

IINC field 78

IIFLG field 79

IIFMT field 78

IILEN field 78

IILTO field 79

IIMMN field 79

IIRFM field 79

IIRSV field 81

IISEC field 80
 IISID field 78
 IITID field 79
 IITST field 80
 IIVER field 78
 INBUF parameter 479
 InhibitEvent attribute 329
 InhibitGet attribute 295
 InhibitPut attribute 296
 InitiationQName attribute 304
 INLEN parameter 479
 INTATR parameter
 MQINQ call 246
 MQSET call 284
 ISS* values 80
 ITI* values 80
 ITS* values 80

L

LN* values 421
 LocalEvent attribute 329
 LT* values 22

M

MaxHandles attribute 329
 MaxMsgLength attribute
 local queue 304
 queue manager 330
 MaxPriority attribute 330
 MaxQDepth attribute 304
 MaxUncommittedMsgs attribute 330
 MD* values 85
 MDACC field 115
 MDAID field 116
 MDAOD field 121
 MDBOC field 112
 MDCID field 111
 MDCSI field 103
 MDENC field 102
 MDEXP field 96
 MDFB field 98
 MDFMT field 104
 MDGID field 121
 MDMFL field 124
 MDMID field 109
 MDMT field 95
 MDOFF field 123
 MDOLN field 128
 MDPAN field 119
 MDPAT field 117
 MDPD field 119
 MDPER field 108
 MDPRI field 107
 MDPT field 120
 MDREP field 85
 MDRM field 113
 MDRQ field 112
 MDSEQ field 123
 MDSID field 85
 MDUID field 114
 MDVER field 85
 ME* values 133
 MECSI field 134
 MEENC field 134
 MEF* values 134

MEFLG field 134
 MEFMT field 134
 MEGID field 134
 MELEN field 134
 MEMFL field 135
 MEOFF field 135
 MEOLN field 135
 MESEQ field 135
 MESID field 133
 message descriptor extension
 structure 131
 message descriptor structure 83
 message order 236, 270, 279
 MEVER field 133
 MF* values 124
 MI* values 111
 MO* values 71
 MQBACK call 203
 MQBEGIN call 207
 MQBO structure 15
 MQBYTE 3
 MQBYTEn 4
 MQCHAR 4
 MQCHARn 4
 MQCIH structure 17
 MQCLOSE call 211
 MQCMIT call 217
 MQCNO structure 31
 MQCONN call 221
 MQCONNX call 227
 MQCONVX call 478
 MQDH structure 37
 MQDISC call 229
 MQDLH structure 43
 MQDXP parameter 479
 MQDXP structure 468
 MQGET call 231
 MQGMO structure 51
 MQHCONN 4
 MQHOBJ 5
 MQIIH structure 77
 MQINQ call 241
 MQLONG 5
 MQMD
 parameter 479
 structure 83
 MQMDE structure 131
 MQOD structure 137
 MQOPEN call 251
 MQOR structure 147
 MQPMO structure 149
 MQPMR structure 165
 MQPUT call 265
 MQPUT1 call 275
 MQRMH structure 169
 MQRR structure 177
 MQSeries for AS/400
 syncpoint considerations with CICS
 for AS/400 339
 syncpoints 338
 MQSeries publications 499
 MQSET call 283
 MQTM structure 179
 MQTMC2 structure 185
 MQWIH structure 189
 MQXCNVC call 473
 MQXQH structure 193

MS* values 305
 MsgDeliverySequence attribute 305
 MSGDSC parameter
 MQGET call 231
 MQPUT call 265
 MQPUT1 call 275
 MT* values 95
 MTK* values 73

N

NameCount attribute 317
 namelist attributes 317
 NamelistDesc attribute 317
 NamelistName attribute 318
 Names attribute 318
 NC* values 317
 notational conventions – RPG
 programming language 10

O

OBJDSC parameter
 MQOPEN call 251
 MQPUT1 call 275
 object descriptor structure 137
 object record structure 147
 OD* values 138
 ODASI field 143
 ODAU field 140
 ODDN field 140
 ODIDC field 141
 ODKDC field 141
 ODMN field 139
 ODON field 139
 ODORO field 142
 ODORP field 143
 ODOT field 138
 ODREC field 141
 ODRMN field 144
 ODRQN field 144
 ODRRO field 142
 ODRRP field 143
 ODSID field 138
 ODUDC field 141
 ODVER field 138
 OII* values 172
 OL* values 22, 128
 OO* values 252, 302
 OpenInputCount attribute 306
 OpenOutputCount attribute 306
 OPTS parameter
 MQCLOSE call 211
 MQOPEN call 251
 MQXCNVC call 474
 ordering of messages 236, 270, 279
 ORMN field 147
 ORON field 147
 OT* values 138
 OUTBUF parameter 480
 OUTLEN parameter 480

P

PDF (Portable Document Format) 502
 PE* values 108, 294
 PerformanceEvent attribute 331

persistence 294
 PF* values 40, 160
 PL* values 331
 Platform attribute 331
 PM* values 150
 PMCT field 158
 PMIDC field 159
 PMKDC field 158
 PMO parameter
 MQPUT call 265
 MQPUT1 call 275
 PMOPT field 150
 PMPRF field 160
 PMPRO field 161
 PMPRP field 163
 PMREC field 160
 PMRMN field 159
 PMRQN field 159
 PMRRO field 162
 PMRRP field 163
 PMSID field 150
 PMTO field 158
 PMUDC field 159
 PMVER field 150
 Portable Document Format (PDF) 502
 PostScript format 502
 PR* values 107
 PRACC field 167
 PRCID field 166
 PRFB field 166
 PRGID field 166
 PRMID field 165
 process definition attributes 319
 ProcessDesc attribute 320
 ProcessName attribute
 local queue 306
 process definition 321
 publications
 MQSeries 499
 related 502
 put-message options structure 149
 put message record structure 165

Q

QA* values
 InhibitGet attribute 296
 InhibitPut attribute 296
 Shareability attribute 310
 QD* values 301
 QDepthHighEvent attribute 307
 QDepthHighLimit attribute 307
 QDepthLowEvent attribute 307
 QDepthLowLimit attribute 308
 QDepthMaxEvent attribute 308
 QDesc attribute 296
 QMgrDesc attribute 331
 QMgrIdentifier attribute 332
 QMgrName attribute 332
 QMNAME parameter 221
 MQCONN call 227
 QName attribute 297
 QRPGLSRC 337
 QServiceInterval attribute 308
 QServiceIntervalEvent attribute 309
 QSIE* values 309
 QT* values 297, 315
 QType attribute 297

queue, dynamic 251
 queue attributes
 alias 315
 common to all queues 293
 local 299
 local definition of remote 313
 model 299
 queue-manager aliasing 313
 queue manager attributes 323

R

RC* values 101, 358
 reason codes
 alphabetic list 357
 numeric list 442
 REASON parameter
 MQBACK call 203
 MQBEGIN call 207
 MQCLOSE call 213
 MQCMIT call 217
 MQCONN call 223
 MQCONN call 227
 MQDISC call 229
 MQGET call 233
 MQINQ call 247
 MQOPEN call 257
 MQPUT call 267
 MQPUT1 call 276
 MQSET call 285
 MQXCNCV call 477
 reference message header structure 169
 RemoteEvent attribute 332
 RemoteQMGrName attribute 313
 RemoteQName attribute 314
 reply queue aliasing 313
 Report field
 using 457
 report message conversion 467
 RepositoryName attribute 332
 RepositoryNameList attribute 332
 response record structure 177
 RetentionInterval attribute 309
 return codes 357
 RL* values 74
 RM* values 170, 171
 RMCSI field 171
 RMDL field 173
 RMDEO field 173
 RMDL field 174
 RMDNL field 174
 RMDNO field 174
 RMDO field 174
 RMDO2 field 175
 RMENC field 171
 RMFLG field 171
 RMFMT field 171
 RMLN field 171
 RMOII field 172
 RMOT field 172
 RMSEL field 172
 RMSEO field 172
 RMSID field 170
 RMSNL field 173
 RMSNO field 173
 RMVER field 170
 RO* values 86
 RPG (ILE) sample programs 341

RPG programming language
 COPY file 8
 notational conventions 10
 structures 10, 337
 RRCC field 177
 RRREA field 177

S

sample programs 341
 browse 344
 echo 348
 get 345
 inquire 349
 preparing and running 343
 put 343
 request 346
 set 351
 trigger monitor 352
 trigger server 352
 using remote queues 353
 using triggering 346
 SCO* values 297
 scope, handles 223, 257
 Scope attribute 297
 SEG* values 73
 SELCNT parameter
 MQINQ call 241
 MQSET call 283
 SELS parameter
 MQINQ call 241
 MQSET call 283
 Shareability attribute 310
 SI* values 144
 SIT* values 143
 softcopy books 502
 SP* values 333
 SRCBUF parameter 476
 SRCCSI parameter 476
 SRCLN parameter 476
 SS* values 73
 StartStopEvent attribute 333
 structures – RPG programming
 language 10, 337
 syncpoint 333
 in CICS for AS/400 applications 339
 with MQSeries for AS/400 338
 SyncPoint attribute 333

T

TC* values 186, 310
 TC2AI field 186
 TC2AT field 186
 TC2ED field 186
 TC2PN field 186
 TC2QMN field 187
 TC2QN field 186
 TC2SID field 186
 TC2TD field 186
 TC2UD field 186
 TC2VER field 186
 terminology xii
 terminology used in this book 487
 TGTBUF parameter 477
 TGTCSI parameter 476
 TGTLN parameter 477

TM* values 180
TMAI field 182
TMAT field 181
TMED field 182
TMPN field 181
TMQN field 180
TMSID field 180
TMTD field 181
TMUD field 182
TMVER field 180
transmission queue header structure 193
trigger message structure 179
TriggerControl attribute 310
TriggerData attribute 310
TriggerDepth attribute 311
triggering 310
TriggerInterval attribute 333
TriggerMsgPriority attribute 311
TriggerType attribute 311
trusted application 32
TT* values 311
type styles in this book xii

U

Uncommitted messages 330
US* values 312
Usage attribute 312
UserData attribute 321

W

WI* values 22, 69, 189
WICSI field 190
WIENC field 190
WIFLG field 190
WIFMT field 190
WILEN field 190
Windows Help 502
WIRSV field 191
WISID field 189
WISNM field 190
WISST field 190
WITOK field 191
WIVER field 189

X

XmitQName attribute 314
XQ* values 196
XQMD field 196
XQRQ field 196
XQRQM field 196
XQSID field 196
XQVER field 196
XR* values 472

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To 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, to this address:

Information Development Department (MP095)
IBM United Kingdom Laboratories
Hursley Park
WINCHESTER,
Hampshire
United Kingdom

- By fax:
 - From outside the U.K., after your international access code use 44-1962-870229
 - From within the U.K., use 01962-870229
- Electronically, use the appropriate network ID:
 - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
 - IBMLink™ : HURSLEY(IDRCF)
 - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5559-00



Spine information:



MQSeries[®] for AS/400[®]

MQSeries for AS/400, V5.1 APR (ILE RPG)

V5.1